مدخل إلى:

علم تصميم البرمجيات

للمؤلف ماكس كانات ألكسندر

المبادرة العربية للترجمة العلمية



مدخل إلى علم تصميم البرمجيَّات

لمؤلفه:

ماكس كانات أليكساندر

نقلته إلى العربيّة:

المبادرة العربيّة للترجمة العلميّة

عن الكتاب الإنجليزيّ:

Code Simplicity

المبادرة العربية للترجمة العلمية

المساهمون

المترجمون:

1. سفيان بغدادي
2. عبد الرحمن مكاوي
3. إسكندر قابة
4. منال لرقم
5. أيوب مكاوي
6. محمد شوقي شطّاح
7. علي عبد العالي
8. هارون خوالديا
9. محمد ماهر عزقول
10. أسامة عبد الرحمن
11. أحمد نور اللّٰه
12. يزيد أسعد
13. مروة الترك
14. ملهم الإبراهيم

المراجعون:

1. حنين محمد أحمد
2. محمد ماهر عزقول
3. مصطفى علي
4. أيوب مكاوي
5. محمد شوقي شطّاح
6. أحمد نور اللّٰه

أبدعت الغلاف: وردان ياسمين

جدول المحتويات

تمهيد

يَكمُن الفرق بين المُبرمِج الجيد والمُبرمِج السيء في الفهم. حيث أنَّ المُبرمِج السيء لا يستوعب جيدًا ما يقوم به، على نقيض المُبرمِج الجيد. صدِّق أو لا تصدق، الأمر بهذه البساطة.

هذا الكتاب موجَّه لمساعدة كل المبرمجين على فهم كيفيَّة تطوير البرمجيّات بشكل إجمالي وجامع حيث أنَّه يمكنك تطبيق ما في الكتاب باستعمال أي لغة برمجة وعلى أي مشروع برمجي من الآن فصاعدًا. يستند الكتاب على قوانين علميّة لتطوير البرمجيات الحاسوبيّة ويعرضها بشكل مُبسَط لدرجة أنَّ أي شخص يمكنه قراءته.

إذا كنت مبرمجًا، ستفهم من خلال هذه القوانين سبب عمل بعض طرق تطوير البرمجيات وعدم عمل أخرى. ستساعدك هذه القوانين أثناء تطوير برمجيّاتك على اتخاذ قراراتك اليوميّة، وستُساعِد فريقك على إجراء نقاشات ذكيّة تودي بهم إلى اتخاذ قرارات واعتماد خطط منطقيّة.

إذا لم تكن مبرمجًا، ولكنك تعمل في صناعة تطوير البرامج الحاسوبيّة بشكل أو بآخر، فقد تجد هذا الكتاب مفيدًا لك لعدة أسباب:

- كونه أداة تعليميّة ممتازة لاستخدامها في تدريب المبرمجين المبتدئين، إضافةً إلى احتوائه على معلوماتٍ مهمة حتى للمبرمجين المتمرسين والخبراء.
- سيمكِّنك من الفهم العميق للأسباب التي تدفع مُهندسي البرمجيّات إلى القيام بأشياء معينة أو السبب وراء وجوب تصميم البرمجيّة الفلانيّة بطريقة معينة.
- سيساعدك على إيصال أفكارك بشكل أوضح لمهندسي البرمجيات الآخرين عبر مساعدتك على فهم المبادئ الأساسية التي يستند عليها المهندسون لاتخاذ قراراتهم.

كل من يعمل في صناعة البرمجيّات الحاسوبيّة ينبغي أن يكون قادرًا على قراءة وفهم محتوى هذا الكتاب، حتى وإن لم يكن لديه الكثير من الخبرة في البرمجة. إذا كان لديك معلومات تقنية في هذا المجال فستتمكن من استيعاب المفاهيم المشروحة بشكل أعمق، و لكن الطابع العام للكتاب لا يحتاج إلى خبرة سابقة في البرمجة لفهمه.

بالرغم من كون هذا الكتاب متمحورًا حول تطوير البرمجيّات، إلا أنّه لا يحتوي حقيقةً على أيّة أكواد برمجيّة. غالبًا ستسأل: كيف يعقل هذا؟ الفكرة من وراء هذا هو أننا نريدك أن تُطبّق محتوى هذا الكتاب على أي مشروع برمجي وبأيّة لغة برمجة موجودة. فلا يجب عليك معرفة لغة بعينها لتستطيع فهم شيء ينطبق على كل لغات البرمجة. ولكنك ستجد في ثنايا هذا الكتاب أمثلة وتطبيقات من الواقع ستساعدك على استيعاب وفهم المبادئ المعروضة وستساعدنا على عرض الفكرة بشكل أوضح.

والأهم من كل هذا هو أنّ الغرض من كتابة هذا الكتاب هو مساعدتك، وعلى إنارة الطريق أمامك، وإرساء النظام والمنطقية والبساطة في مجال تطوير البرمجيّات. أتمنى أن تستمتع بقراءته وأن تتمكّن من تحسين حياتك وبرمجيّاتك من خلاله.

الفصل الأوَّل: مقدمة

أحدثت الحواسيب تغييرًا اجتماعيًا كبيرًا، ذلك لأنَّها تسمح لنا بإنجاز عمل أكثر بأفراد أقل، وهنا تكمن قيمة الحواسيب: إنجاز عمل أكثر بسرعة أكبر.

المشكلة الأساسيّة في الحواسيب هي أنها تتعطل دائمًا، ولو تعطل أي شيء في منزلك كما تتعطل الحواسيب لأعدته. يعاني أغلب الناس في المجتمعات المُتحضِّرة من تعطل أو سوء سلوك الحواسيب مرَّة يوميًا على الأقل، والذي هو بالأمر السيء.

ما مشكلة الحواسيب؟

لماذا تتعطل الحواسيب كثيرًا؟ بالنسبة للبرمجيَّات، يوجد سبب واحد فقط يُسبِّب تعطلها، وهو البرمجة السيئة. يلقي بعض المُستخدمين اللوم على الإدارة وبعضهم على العملاء، لكنَّ التحريات تظهر أنَّ أصل المشكلة يكون دومًا في البرمجة نفسها.

ماذا نعني بقولنا "البرمجة السيئة"؟ وكيف يُبرمِج المبرمجون الذين يعتبرون أشخاصًا أذكياء جدًّا ومنطقيين بشكل سيء؟

الأمر كله متعلق بالتعقيد.

من الجائِز أنَّ الحاسوب أكثر الأجهزة الإلكترونيَّة التي يمكننا صناعتها اليوم تعقيدًا. حيث يمكنه إجراء مليارات العمليات بالثانية. كما يحتوي مئات الملايين من الأجزاء الإلكترونية الدقيقة حتى يعمل بشكل صحيح.

البرنامج المُبرمَج على الحاسوب مُعقَّد بشكل مماثل، فمثلًا عندما بُرمِجَ مايكروسوفت ويندوز 2000 كان يُعد واحدًا من أكبر البرامج على الإطلاق والذي حوى نحو 30 مليون سطر آنذاك. كتابة هذا القدر من الأكواد يعادل كتابة كتاب من 200,000,000 كلمة (ما يعادل 5 مرّات ضعف حجم موسوعة بريتانيكا).

يمكن لتعقيد برنامج ما أن يكون مربكًا لعدم كونه شيئًا ملموسًا. عندما ينهار البرنامج، لن تجد شيئًا صلبًا تستطيع أن تتفحصه بكلتا يديك أو تنظر إليه. كل شيء مُجرَّد، مما قد يجعل التعامل معه صعبًا. برنامج الحاسوب العادي حقيقةً مُعقَّد جدًّا لدرجة أنَّه لا يمكن لأي شخص أن يفهم طريقة عمل كل الأكواد البرمجية فيه في مجملها. كلما ازداد حجم البرامج، كلما ازدادت درجة تعقيدها.

ولذلك صارت البرمجة بمثابة عملية **تبسيط للتعقيد**، وإلا فلن يكون من الممكن مواصلة العمل على برنامج ما بعد بلوغه درجة معينة من التعقيد. يجب على الأجزاء المعقدة من البرنامج أن تكون منظمة بطريقة مُبسطة حتى يتسنى للمبرمج العمل عليها دون الحاجة إلى امتلاك قدرات عقلية خارقة للطبيعة.

هذا هو الفن والموهبة المطلوبان في البرمجة: تبسيط التعقيد.

"المبرمج السيء" هو فقط شخص يفشل في تبسيط التعقيد. يحدث هذا كثيرًا عندما يعتقد الناس أنهم يبسطون تعقيد الكتابة بلغة برمجية (وهو قطعًا أمر معقد بطبيعته) بكتابة أكواد "تعمل وحسب"، دون التفكير في تبسيط التعقيد من أجل المبرمجين الآخرين.

الأمر، إلى حدٍ ما، كما يلي: تخيّل مهندسًا بحاجة إلى شيءٍ ما ليدق به مسمارًا في الأرض. يخترع هذا المهندس جهازًا مصنوعًا من بكراتٍ وسلاسل ومغناطيسٍ كبير من أجل هذه المهمة. ستعتقد على الأرجح أنَّ هذا سخيف للغاية.

والآن تخيّل أن يقول لك أحدهم: "أحتاج أكوادًا يمكنني استعمالها في **أي برنامج، وفي أي مكان**. كما يمكنها أن تؤمن الاتصال بين أي حاسوبين، عبر أي وسيلة يمكن تخيلها". بالتأكيد تبسيط هذا البرنامج سيكون أكثر صعوبة. ولذلك بعض المبرمجين (ربما **جميع** المبرمجين) في هذه الحالة سيأتون بحل يستخدم ما يكافئ السلاسل والبكرات والمغناطيس الكبير، وسيكون بالكاد مفهومًا للآخرين. هذا لا يعني أنَّهم غير منطقيين، أم أنَّ بهم خطبًا. ولكنهم عندما يواجهون مهمة صعبة فإنهم سيفعلون ما باستطاعتهم

في الإطار الزمني الضيق الذي يملكونه. ما يصنعونه **سيعمل**، هذا ما يهمهم. ذلك ما يريده رب عملهم، وما يبدو أنَّ زبائنهم يريدونه كذلك.

لكن بشكل أو بآخر، سيكونون قد فشلوا في تبسيط التعقيد. سيمررون بعد ذلك هذا الجهاز إلى مبرمج آخر ليزيد التعقيد باستعماله كجزء من جهازه. كلما زاد عدد الأشخاص الذين لا يفعلون شيئًا لتبسيط تعقيد البرنامج، كلما صار البرنامج مُبهَمًا أكثر.

عندما يقترب البرنامج من التعقيد اللانهائي، يصبح من المستحيل العثور على جميع المشاكل فيه. يبلغ ثمن الطائرات النفاثة ملايين أو مليارات الدولارات لأنَّها **قريبة** من هذا التعقيد و"مُنقَّحة". لكن أغلب البرمجيات العادية يتراوح ثمنها ما بين 50 و 100 دولار، وبهذا الثمن لن يكون لأحد الوقت أو الموارد اللازمة للتنقيب عن جميع المشاكل في نظام لانهائيِّ التعقيد وتصحيحها.

لذلك يجب على المبرمج الجيد أن يبذل كل ما بوسعه ليُبسِّط ما يكتبه للمبرمجين الآخرين. يصنع المبرمج الجيد أشياء سهلة الفهم، وهكذا يُسهِّل التَخلُّص من العلل.

فكرة البساطة هذه تُفهَم أحيانًا بطريقة خاطئة. حيث تُفهَم أنَّ البرنامج لا ينبغي أن يحوي الكثير من الأكواد أو لا يجب أن يستعمل تقنياتٍ متقدمة، ولكن هذا ليس صحيحًا.

كثرة الأكواد تؤدي أحيانًا إلى البساطة، ما يُكلِّف المزيد من القراءة والكتابة، والذي ليس بالأمر الجَلل. ويجب عليك عندها كذلك أن تتأكد من امتلاكك لتوثيقٍ قصير يشرح الكود، ولكن هذا أيضًا جزء من تبسيط التعقيد. يؤدي عادةً كذلك استخدام المزيد من التقنيات المتقدمة إلى المزيد من البساطة، رغم أنه يجب عليك تعلمها أولًا، والذي قد يكون فيه بعض العناء.

يعتقد بعض الناس أنَّ الكتابة بطريقة بسيطة تأخذ وقتًا أكبر من كتابة شيء بعجلة لإكمال العمل. ولكن حقيقةً إمضاء وقت أكثر بقليل في كتابة كود بسيط يتضح أنَّه أسرع من كتابة العديد من الأكواد بعجلة في البداية ثم إمضاء الكثير من الوقت في محاولة فهمها. هذا تبسيط كبير للمشكلة، لكن هذا ما يريه

تاريخ مجال البرمجة. العديد من البرامج الرائعة راكدة في تطورها على مرّ السنين فقط لأنَّها استغرقت وقتًا طويلًا لإضافة ميزات إلى الوحوش المعقدة التي أصبحت عليها الآن.

و لهذا السبب تتعطَّل الحواسيب في العديد من الأوقات؛ لأنَّ في معظم برامجها الرئيسيّة، العديد من مبرمجي الفريق فشلوا في تبسيط تعقيد الأجزاء التي كانوا يكتبونها. نعم، هذا صعب، لكن هو لا شيء مقارنة بالصعوبة اللامُتناهية التي يواجهها المستخدمون عند استعمالهم لأنظمة معقدة وسهلة التعطُّل مُصممَّة من مبرمجين فشلوا في تبسيطها.

ما هو البرنامج؟

للبرنامج الحاسوبي تعريفان متمايزان:

1. سلسلة من التعليمات المُمرَّرة للحاسوب.
2. الإجراءات التي يتخذها الحاسوب كنتيجة للتعليمات المُمرَّرة.

التعريف الأوَّل هو ما يراه المبرمجون عند كتابة البرنامج، بينما التعريف الثاني هو ما يراه المستخدمون عند استخدامهم للبرنامج. يخبر المُبرمِج الحاسوب "اعرض ماعِزًا على الشاشة" والذي يُمثِّل التعريف الأوَّل، أي بعض التعليمات والأوامر. يستخدم الحاسوب الكهرباء لعرض الماعِز على الشاشة، والذي هو التعريف الثاني، أي الإجراءات المتخذة من طرف الحاسوب. سيقول كل من المبرمج والمستخدم هنا أنَّهما يستعملان "برنامج حاسوبي"، لكن تجربتهما مختلفة تمامًا. المبرمجون يتعاملون مع الكلمات والرموز، بينما يرى المستخدمون النتيجة النهائية فقط (الإجراءات المُتخذة).

برنامج الحاسوب هو كل هذه الأشياء: التعليمات التي يكتبها المبرمج والإجراءات التي يتخذها الحاسوب. الغاية من كتابة التعليمات هي التسبُّب في حدوث هذه الإجراءات، فبدون الإجراءات لن يكون هناك سبب لكتابة التعليمات.

الأمر أشبه بكتابة قائمة مشتريات في الحياة العملية. يمكننا أن نعتبر القائمة مجموعة من التعليمات حول ما يلزم شراؤه من المتجر. وبالتالي لا معنى لكتابة هذه التعليمات دون الذهاب فعليًا إلى المتجر لشراء الحاجيات. لا بدّ للتعليمات أن تُسببِّ حدوث شيء ما.

غير أنَّ هناك فرقًا كبيرًا بين كتابة قائمة مشتريات وكتابة برنامج حاسوبي. كون القائمة غير مرتبة سيتسبب في إبطاء عملية الشراء وحسب. لكن إذا كانت الأكواد التي كتبتها غير مرتبة، فإنَّ الوصول إلى أهدافك البرمجية سيصبح صعبًا للغاية. لماذا؟ قوائم المشتريات قصيرة وبسيطة، كما يمكنك التخلُّص منها حال الانتهاء من استعمالها. لكن برامج الحاسوب كبيرة ومعقدة، كما أنَّك في الغالب ستحتاج للاحتفاظ بها لسنوات أو لعقود عدة. ولذلك، بينما لا تتسبَّب قائمة مشتريات بسيطة وقصيرة بالكثير من الصعوبات مهما كانت غير مرتبة، يمكن لأكواد الحاسوب غير المرتبة أن تتسبب بقدر عظيم من الصعوبة.

إضافةً إلى ذلك، لا يوجد أي مجال ترتبط فيه النتائج بسلسلة التعليمات المُسبِّبة لها ارتباطًا وثيقًا مثلما هو الحال في مجال تطوير البرمجيَّات. حيث يكتب الأشخاص في المجالات الأخرى تعليمات ثم يسلمونها لأشخاص آخرين، لينتظروا في الغالب وقتًا طويلًا قبل أن يروها تُنفَّذ. فمثلًا، عندما تُصمِّم مهندسة معمارية منزلًا، فإنَّها تكتب مجموعة من التعليمات: أي المخططات. تمر هذه الأخيرة عبر أشخاص كُثُر في وقت زمني طويل جدًّا قبل أن تتحوَّل إلى بناء ملموس. البناء النهائي هو نتيجة ترجمة كل هؤلاء الأشخاص لتعليمات المهندسة المعمارية على أرض الواقع. لكن على النقيض من ذلك، لا يوجد شخص وسيط بيننا وبين الحاسوب عند كتابتنا للأكواد. النتيجة مطابقة **تمامًا** لما طلبت التعليمات من الحاسوب القيام به، من غير نقاش. جودة النتيجة النهائية في هذه الحالة تعتمد كليًّا على جودة الآلة وجودة أفكارنا وجودة الأكواد التي كتبناها.

من بين هذه العوامل الثلاثة، جودة الأكواد هي أكبر مشكلة تواجه مشاريع البرمجة اليوم. لهذا السبب، ومن أجل النقاط المذكورة أعلاه، يهتم الجزء الأكبر من هذا الكتاب بتحسين جودة الأكواد. سنتطرق

لجودة الأفكار والآلات في أماكن قليلة من الكتاب، لكن أغلب التركيز سينصب على تحسين بنية وجودة التعليمات التي تعطيها للآلة.

تجدر الإشارة إلى أنَّه من السهل جدًّا نسيان هدفنا الأساسي الذي نرغب في تحقيقه عندما نقضي وقتًا طويلًا في الحديث عن الأكواد، ألا وهو تحقيق **نتيجة** أفضل. لا شيء في هذا الكتاب يتسامح مع النتائج الضعيفة، ما يجعلنا نُركِّز على تحسين الأكواد هو **أنَّ تحسينها أهم مشكلة يجب أن نحلها حتى نتمكن من تحسين النتيجة**.

وبالتالي نحن بأمس الحاجة إلى علمٍ ما لِتحسين جودة الأكواد.

الفصل الثاني: العلم المفقود

يتمحور هذا الكتاب في أغلبه حول ما يُدعى "بتصميم البرمجيات". ربما سمعت عن هذا المصطلح من قبل، وربما قرأت عنه بعض الكتب. لكن دعنا نتناوله من منظور حديث، مع تعريفات جديدة ودقيقة. نحن نعرف ماذا يعني "برنامج"، إذًا ما علينا تعريفه الآن هو "التصميم":

الفعل صمَّمَ:

1. أعدَّ خطّةً للبناء.

مثال: سيُصمِّم المُهندس جسرًا هذا الشهر وسيبنيه الشهر المقبل.

الاسم تصميم:

1. خطة أُنجِزَت لبناء معين لم يُشيَّد بعد.

مثال: جاء المُهندس بتصميم لجسرٍ سيشرع في بنائه الشهر المقبل.

2. خطة يتبعها بناء قائم.

مثال: يتبع ذلك الجسر هناك تصميمًا جيدًا.

كل هذه التعريفات يمكن تطبيقها عندما نتحدث عن " تصميم البرمجيات":

- حينما "نُصمِّم البرمجيّة" (ونُصمِّم هنا تتبع تعريف الفعل "صمَّم") هنالك العديد من الأمور نخطط لها (هيكلة الكود، التقنيات المستعملة...إلخ)، وهناك الكثير من القرارات التقنية التي يجب اتخاذها. نتخذ غالبًا هذه القرارات في أذهاننا فقط، ولكن في بعض الأحيان نكتب أيضًا الخطط وننجز بعض المخططات البيانية.

- حينما ننتهي من ذلك تكون النتيجة تصميم البرمجيَّة ("تصميم" هنا تتبّع التعريف الأوَّل للاسم "تصميم") التي نعمل عليها. من الممكن أن يكون هذا التصميم على شكل توثيقٍ مكتوب، أو مجرد مجموعة من القرارات التي اتخذناها في أذهاننا.
- الكود الذي يُكتَب له أيضًا تصميم (التعريف الثاني للتصميم) والذي يُعتبر البنية أو الخطة التي يتبعها هذا الكود. هناك بعض الأكواد التي ليس لها بنية واضحة على الإطلاق، هذه الأخيرة ليس لها تصميم، ما يعني أنَّه لا يوجد لها خطّة مُحدَّدة من طرف المُبرمِج. ما بين " التصميم" و"غياب التصميم" هناك أيضًا حالات أخرى، مثل: "تصميم جزئي" و "تصاميم متعارضة موجودة في جزء واحد من الكود" و"تصميم يكاد يكتمل" وغيرها. يمكن أيضًا أن يكون وجود التصاميم أسوأ من غيابها، فمثلًا كود كُتب عمدًا على نحو معقد وغير منظّم سيكون ذو تصميم سيء للغاية.

علم تصميم البرمجيات هو علم وَضْع الخطط والقرارات المتعلقة بالبرمجيات. يساعد هذا العلم على الإجابة عن أسئلة مثل:

- كيف ينبغي أن تكون بنية الكود في برنامجنا؟
- أيهما أهم: التركيز على تصميم برنامج أداؤه سريع أم برنامج كوده سهل القراءة؟
- أي لغة برمجة يجدر بنا استخدامها لمشروعنا؟

علم تصميم البرمجيات لا يجيب عن أسئلة مثل:

- كيف ينبغي أن تكون هيكلة الشركة؟
- متى ينبغي أن تكون اجتماعات الفريق؟
- أي أوقات في اليوم يجب أن يعمل فيها المبرمجون؟
- كيف يُقاس أداء المبرمِجين؟

هذه ليست قرارات بشأن منتجك البرمجيّ، وإنّما هي قرارات تتعلق بك أو بمؤسستك. من المؤكّد أنّ اتخاذ قرارات كهذه بشكل صحيح أمر بالغ الأهميَّة، فهناك الكثير من المشاريع التي فشلت بسبب سوء الإدارة، ولكن ليس هذا محور كتابنا. يتحدث هذا الكتاب عن كيفيَّة اتخاذ القرارات التقنية الصحيحة بشأن برمجيّاتك.

يندرج أي شيء يتعلق بمعمارية نظامك أو القرارات التقنية التي تقوم بها أثناء إنشائها ضمن فئة: "تصميم البرمجيات".

كل مبرمج مصمِّم

يشارك كل مبرمج يعمل في مشروع برمجي في التصميم. المطوّر الرئيسي هو المسؤول عن تصميم البنية العامّة للبرنامج. كبار المبرمجين هم المسؤولون عن تصميم الأجزاء الكبيرة التي يشرفون عليها، والمبرمجون المبتدئون مسؤولون عن أجزائهم من البرنامج حتى ولو كانت هذه الأجزاء ببساطة العمل على جزء واحد من ملف واحد. قد تدخل حتى اعتبارات تصميميّة في كتابة سطر برمجيّ واحد.

حتى ولو برمجتَ كل شيء بنفسك، تبقى عملية التصميم هذه موجودة. أحيانًا قد تتخذ قرارًا على الفور قبل أن تضغط أصابعك على لوحة المفاتيح وتنتهي العملية، وأحيانًا ستفكر في كيفيَّة كتابة البرنامج وأنت على سرير نومك.

كل مَنْ يكتب البرمجيّات يُعدّ مُصمِّمًا، وكل فرد في الفريق مسؤول عن التأكُّد من أنَّ كوده مُصمَّم جيدًا، ولا أحد بإمكانه أن يتجاهل التصميم على أي مستوى.

ومع ذلك، لا ينبغي أن تكون عمليَّة التصميم عمليَّة ديموقراطيَّة، **ولا يجب** أن تتم بواسطة لجنة مُخصّصة؛ لأنَّ النتيجة غالبًا ستكون تصميمًا سيئًا. بل يجب أن يكون لدى جميع المطورين السلطة في اتخاذ قرارات التصميم المناسبة فيما يتعلق بأجزائهم، والقرارات الضعيفة والغير موفقة تُستبدَل من

طرف المطوِّر الخبير أو المطوِّر الرئيسي الذي يملك حق النقض في قرارات المصممين الآخرين[1]. المهم أن تكون مسؤولية تصميم الكود تقع على عاتق الأشخاص الذين يعملون فعليًّا عليه.

يجب أن يكـون المصـمم دائمًا على اسـتعداد للاسـتماع إلى الاقتراحـات والتعليقـات والتقييمـات؛ لأنَّ المبرمجين هم عـادةً أشخاص أذكيـاء لـديهم أفكـار جيـدة. ولكن بعـد النظـر في جميـع الاعتبـارات، يجب اتخاذ القرار فرديًّا لا جماعيًّا.

علم تصميم البرمجيّات

تصميم البرمجيّات، كما يُمارَس في العالم اليوم، ليس علمًا. فما هو العلم؟ تعريف القاموس للعلم معقّد بعض الشيء، لكن في الأساس لكي يكون موضوع ما علمًا عليه اجتياز اختبارات معينة:

- ينبغي أن يحـوي العلم على معرفـة مَجموعـة مُكوَّنـة من مجموعـة حقـائق لا آراء، وهـذه الحقـائق ينبغي أن تكون موضوعة معًا في مكانٍ ما (ككتاب مثلًا).
- هذه المعرفة لا بدّ أن يكون لها نوع من التنظيم. فينبغي أن توضع في فئات، وأن تكون الأجزاء المتنوعة مرتبطة بشكل صحيح ببعضها البعض من حيث الأهمية ..إلخ.
- يجب أن يحتوي العلم على حقاق عامة أو قوانين أساسية.
- ينبغي على العلم أن يخـبرك كيفيَّة القيـام بشـيء في العـالم المـادي، بحيث أن يكـون بطريقـة أو بأخرى قابلًا للتطبيق في الحياة الواقعيّة.

1. حاول أن تشرح للمبرمجين سبب استبدالك لقرار إن فعلت، وأظهر لهم سبب أو كيفيَّة كون قرارك هو الأنسب والأفضل. سيُقِّلل شرحك هذا عبر الوقت القرارات التي ستستبدلها مُستقبلًا. بعض المبرمجين لا يتعلمون أبدًا، وإذا كان منهم مَنْ يستمر في اتخـاذ العديـد من القـرارات السـيئة في التصـميم، على الـرغم من الشـرح المتواصـل بعـد عـدة شـهور أو سـنوات، فيجب استبعاده من فريقك. لكن أغلب المبرمجين هم أشخاص أذكياء يتعلمون سريعًا، لذلك نادرًا ما سيكون هذا مصدرًا للقلق.

- يُكتشَف العلم عادةً ويُثبَت من خلال منهجيّة علميّة، تنطوي على ملاحظة العالم المادي ووضع نظرية حول كيفيَّة عمل هذا الأخير وإجراء تجارب للتحقق من النظرية وإظهار أنَّ التجربة نفسها تعمل في كل مكان؛ لإثبات أنَّ النظرية عامة وليست مجرد مصادفة أو أنه شيء يعمل عندك فقط.

في عالم البرمجيات، لدينا الكثير من المعرفة المجمّعة والمنظّمة في الكتب. ولكن مع توفر جميع الأجزاء اللازمة لتكوين علم، إلا أننا نفتقد الجزء الأكثر أهمية: قوانين محددة بوضوح وصلبة ولا تتزعزع أبدًا.

مطوري البرمجيات الخبراء يعرفون ما هو الشيء الصحيح الذي ينبغي عمله، ولكن لماذا هذا هو الشيء الصحيح؟ ما الذي يجعل بعض القرارات صحيحة وبعض القرارات خاطئة؟ ما هي القوانين الأساسية لتصميم البرمجيات؟

يضع هذا الكتاب مجموعة من التعاريف والحقائق والقواعد والقوانين في تطوير البرمجيات، والتي تتركز في الغالب على تصميم البرمجيات.

ما الفرق بين الحقيقة والقاعدة والتعريف والقانون؟

- تخبرك التعريفات عن ماهيّة الشيء وكيفيَّة استعماله.
- الحقائق هي مجرد عبارات صحيحة عن شيء ما. أيّ معلومة صحيحة هي حقيقة.
- القواعد هي عبارات تعطيك توجيهات صحيحة وتغطي شيئًا محددًا وتساعد في توجيه القرار، ولكن لا تساعدك بالضرورة على التنبؤ بما سيحدث في المستقبل أو على تبيّن حقائق أخرى. عادةً ما تخبرك القواعد كذلك ما إذا كان يجب اتخاذ بعض الإجراءات أم لا.
- القوانين هي الحقائق التي تكون دائمًا صحيحة، و تغطي مجالًا واسعًا من المعرفة. كما تساعدك على اكتشاف حقائق مهمة أخرى وتُمكِّنُك من التنبؤ بما سيحدث مستقبلًا.

من بين كل هذا، القوانين هي الأكثر أهمية، وهي مبيّنة بوضوح في هذا الكتاب. إذا لم تكن متأكدًا من الفئة التي تقع فيها بعض المعلومات، فستجد في "الملحق ب" كل معلومة رئيسية في الكتاب، موضّحة كقانون أو قاعدة أو تعريف أو مجرد حقيقة عادية.

عندما تقرأ بعض هذه التعريفات أو القوانين أو القواعد أو الحقائق، قد تقول لنفسك: "هذا واضح، لقد كنت أعرف ذلك من قبل!"، وذلك متوقع إذا كنت شغّالًا في مجال تطوير البرمجيات لفترة طويلة. إذا مررت بهذا، اسأل نفسك ما يلي:

- هل كنتُ أعرف أنَّ هذه المعلومة مبرهنة؟
- هل كنتُ أعرف مدى أهميتها؟
- هل كان بإمكاني توصيلها بوضوح إلى شخص آخر حتى يفهمها كاملةً؟
- هل فهمتُ كيفيَّة ارتباطها ببيانات أخرى في مجال تطوير البرمجيّات؟

إذا كان بإمكانك أن تقول "نعم" لبعض هذه الأسئلة في حين كنت قد قلت سابقًا "لا" أو "ربما"، عندئذٍ ستكون قد اكتسبت نوعًا معينًا من الفهم. هذا الفهم جزء كبير مما يميِّز العلم من مُجرَّد مجموعة من الأفكار.

قد لا يكون هذا بالطبع علمًا مثاليًا كاملًا بَعْد، فهناك دائمًا شيء آخر لاكتشافه في الكون، وأشياء لمعرفتها أكثر في أي مجال، وفي بعض الأحيان هناك تصحيحات يجب إدخالها على القوانين الأساسية عندما تُكتشف بيانات تُظهِر حقائق جديدة. ولكن هذه مجرَّد نقطة للانطلاق. هذا شيء يُمكننا البناء عليه مستقبلًا: هذه مجموعة حقيقية من القوانين والحقائق لبناء البرمجيّات.

حتى ولو ثَبُت خطأ أجزاء من هذا الكتاب في يومٍ من الأيام وطُوِّر علم أفضل، من المهم أن تبقى حقيقة واحدة واضحة: تصميم البرمجيّات يمكن أن يكون علمًا، وإنّه ليس سرًّا أبديًّا رهنًا برأي كل مبرمج أو كل خبير استشاري يريد أن يربح بضعة دولارات من بيع "طريقة جديدة" في تصميم البرمجيات.

هناك قوانين يُمكِن ويمكنك معرفتها. هذه القوانين أبدية وغير مُتغيّرة وصحيحة في جوهرها وتعمل.

فيما يتعلَّق بصحّة بعض القوانين المذكورة في الكتاب أو خطئها، فهناك المئات من الأمثلة والتجارب التي يمكن الاستشهاد بها في الإثبات، ولكن في النهاية، عليك أن تقرر بنفسك. اختبر القوانين وتحقَّق مما إذا كان بإمكانك التفكير في أيّ من الحقائق العامة حول تطوير البرمجيات بشكل أوسع أو أكثر أساسية. إذا بدا لك أي شيء أو وجدت مشكلة مع القانون، توجَّه إلى موقع الكتاب لمعرفة كيفيَّة الاتصال بالمؤلف وعرض مساهماتك أو أسئلتك. أي تحسينات أخرى في هذ الموضوع سوف تفيد الجميع، طالما أنَّ هذه التحسينات صحيحة فعلًا، أو طالما كانت قوانين أساسية أو قواعد جديدة لتصميم البرمجيات.

لماذا لم يكن هناك علم لتصميم البرمجيات؟

ربما كنت ترغب في معرفة سبب عدم وجود علم لتصميم البرمجيّات قبل صدور هذا الكتاب، فلقد كتبنا البرمجيات لعقود وعقود للآن.

إنَّها قصة مثيرة للاهتمام. في ما يلي بعض المعلومات والخلفيات التي قد تساعدك على فهم كيف وصلنا إلى حد بعيد مع أجهزة الحاسوب دون تطوير علم تصميم البرمجيّات.

ما نعرفه اليوم على أنّه "حاسوب" قد بدأ في أذهان علماء الرياضيات كأدوات مجردة بحتة، أو أفكار حول كيفيَّة حل مشاكل الرياضيات باستخدام الآلات بدلاً من العقل. هؤلاء الرياضيين هم الأشخاص الذين قد نعتبرهم مؤسسي علم الحاسوب، وهو الدراسة الرياضيَّة لمعالجة المعلومات، وليست دراسة برمجة الحاسوب كما يعتقدها البعض. بُنيَت الحواسيب الأولى من طرف علماء الحاسوب هؤلاء تحت

إشراف مهندسي إلكترونيات ذوي مهارات عالية وكانت تُشغَّل من قبل مشتغلين مدربين تدريبًا عاليًّا وفي بيئات مراقبة بإحكام. كانت هذه الأجهزة موجّهة خصيصًا للجهات التي طلبتها وغالبًا كانت موجهة للحكومات (لتوجيه الصواريخ وفك الأكواد). وكانت هناك نسخة واحدة أو نسختين فقط من أي نموذج.

بعدها جاءت UNIVAC وهي أوّل الحواسيب التجارية في العالم. عند هذه النقطة، لم يكن يوجد بالعالم إلا مجموعة من علماء الرياضيات النظرية المتقدمة. عندما بدأت الحواسيب تصبح متاحة للجميع، كان من الواضح أنَّه من غير الممكن شحن رياضي مع كل حاسوب. وبالرغم أنَّ بعض المنظمات، مثل مكتب الإحصاء في الولايات المتحدة (أحد أول المستلمين لحواسيب UNIVAC)، كانت تملك عدة عاملين على درجة عالية من التدريب من أجل آلاتها، فإنَّ منظمات أخرى لا شك حصلت على آلاتها ثم قالت: "(بيل) من قسم المحاسبة، هذا جهازك! اقرأ دليل الاستعمال وحاول تشغيله!". عندها توجَّه (بيل) إلى هذه الآلة المُعقَّدة مُقدِمًا أحسن ما عنده لتشغيلها.

بذلك أصبح (بيل) من أوائل "المبرمجين العاملين". ربّما يكون قد درس الرياضيات في المدرسة، لكنه على الأغلب لم يدرس ذلك النوع من النظريات المتقدمة المطلوبة لفهم وتصميم آلة بحد ذاتها. مع ذلك كان قادرًا على قراءة الدليل وفهمه، ومع المحاولة والخطأ، استطاع أن يجعل الآلة تفعل ما يريده. وطبعًا، كلما ازداد عدد الحواسيب التجارية المشحونة، كلما ازداد عدد العاملين غير المؤهلين مثل (بيل). الأغلبية العظمى من المبرمجين انتهى بهم الأمر مثل (بيل)، وإن كان هناك شيء واحد إضافي قد حصل عليه (بيل) فهو ضغط العمل. تلقى طلبات عديدة من الإدارة مثل: "أنجز هذه المهمة الآن!" وقيل له: "لا يهمنا كيف تنجز ذلك، أنجزه فقط!". توصَّل (بيل) الى طريقة تُمكِّنه من جعل الأشياء تعمل وفقًا للدليل، حتى ولو انهارت كل ساعتين.

حصل (بيل) في نهاية الأمر على فريق كامل من المبرمجين ليعملوا معه. كان عليه أن يكتشف كيفيَّة تصميم نظام وتقسيم المهام بين أشخاص مختلفين. تطوّر فن البرمجة العملية بشكل كامل عضويًّا. كان

الأمر أشبه بتعليم طلبة الكلية لأنفسهم كيفيِّة الطبخ أكثر من صناعة مهندسي ناسا لمركبة فضائية.

عند هذه المرحلة، أصبح هناك نظام تطوير برمجيات مختلط، وهو معقد جدًا ويصعب إدارته، لكن الجميع يجدون مخرجًا بطريقة ما. ثم جاء كتاب "أسطورة رجل بالشهر" (The Mythical Man Month) لـ(فريد بروكس)، والذي نظر إلى عمليَّة تطوير البرمجيَّات في مشروع حقيقي وأشار إلى بعض الحقائق حول هذا الموضوع. أكثرها شهرة: أنَّ إضافة مبرمجين الى مشروع برمجيَّات مُتأخِّر يزيده تأخرًا. هو لم يأتِ بعلمٍ كامل، لكنَّه قام بعدة ملاحظات جيدة حول برمجة وإدارة تطوير البرمجيَّات.

بعد ذلك جاءت موجة من منهجيات تطوير البرمجيَّات، مثل: العمليَّة الموحدة لراشيونال ونموذج نضج القدرة والتطوير السريع للبرمجيات وغيرها من المنهجيات الأخرى. لم تدّعِ أي منها أنها علم، كانت مجرد منهجيات لإدراة تعقيد تطوير البرمجيّات.

وهذا أساسًا ما أوصلنا إلى ما نحن عليه اليوم، الكثير من المنهجيِّات لكن دون علم حقيقي. هناك في الواقع علمان مفقودان: علم إدارة البرمجيَّات وعلم تصميم البرمجيَّات.

علم إدارة البرمجيات سيُخبرنا بأمور مثل: كيف نُقسِّم العمل بين المبرمجين وكيف نُجدول الإصدارات وكيفيَّة تقدير الوقت الذي سيستغرقه كل عمل وهكذا دواليك.

يجري العمل بهذا العلم بنشاط وهو مُعالَج من مختلف المنهجيات المذكورة أعلاه. حقيقة وجود آراء متناقضة وصالحة في نفس الوقت في هذا المجال تشير الى أنَّ القوانين الأساسيَّة لإدارة البرمجيات لم تُوضَع بعد، ولكن على الأقل يوجد اهتمام بهذه المشكلة.

بينما، في الجهة المقابلة، علم تطوير البرمجيات يلقى اهتمامًا قليلًا في العالم التطبيقي للبرمجة. عدد قليل جدًّا من الناس تعلَّموا في المدرسة أنَّه من الممكن أن يكون هناك علم لتصميم البرمجيات في المستقبل. ولكن قيل لهم بدلًا من هذا: "هكذا تعمل لغة البرمجة هذه، الآن اذهبوا واصنعوا برنامجًا ما!".

هذا الكتاب وُجِدَ لملء تلك الفجوة.

العلم المُقدَّم هنا ليس علم الحاسوب. تلك دراسة رياضيّة، في حين أنَّ هذا الكتاب يحتوي على مقدمة علم لـ "المبرمجين العاملين"، أي مجموعة من القوانين الأساسيَّة والقواعد التي تُتبع عند كتابة برنامج بــأي لغة. هــذا علم موثــوق مثــل الفيزيــاء أو الكيميــاء يخــبرك كيــف تقــوم بإنشــاء التطبيقــات. هناك من قال أن علمًا مثل هذا غير ممكن، وأنَّ تصميم البرمجيَّات متغيِّر جدًّا بحيث لا يمكن وصفه من خلال قوانين أساسيَّة بسيطة، وأنَّ الأمر ككل هو مجرد مجموعة من الآراء. هناك أيضًا من قالوا من قبل أنَّ فهم الكون المادي يعد مستحيلا لأنَّه: "خلق الله والله لا سبيل لمعرفته"، ومع ذلك استطعنا اكتشاف علـوم الكـون المـادي. إذًا مـا لم تكن تـؤمن أنَّ الحواسـيب لا سـبيل لمعرفتهـا، فـإنَّ إنشـاء علم لتصـميم البرمجيّات يجب أن يكون ممكنًا.

هناك أيضا خرافة تقول أنَّ البرمجة شكل من أشكال الفنون، تخضع كليًّا لرغبات المبرمج. صحيح أنَّ هنـاك بعض الفن المُتضـمِّن في تطـبيق أي علم من العلـوم، لكن يتعيّن أن يكـون هنـاك علم حـتى يُطبَّق، وحاليًّا لا يوجد.

المصدر الأوَّل للتعقيد في أي برنامج حقيقةً هو غياب هذا العلم. في الواقع لو كان المبرمجون يملكون علمًا لتبسيط البرامج، لما كان هناك الكثير من التعقيد تقريبًا، ولما كُنَّا بحاجة إلى عمليات جنونيَّة لإدارة هذا التعقيد.

الفصل الثالث: الدافع وراء تصميم البرمجيات

عندما نبرمج يجب أن نعرف لِمَ نقوم بذلك وما الهدف المرجو منه، فهل نستطيع أن نستخلص غاية لكل البرمجيات؟ لو أجبنا على هذا السؤال لكُنّا قد عرفنا أين نتجه وذلك كان سيغير علم تصميم البرمجيات بأكمله.

في الحقيقة يوجد غاية وحيدة لكل البرمجيات، وهي:

مساعدة الناس[2].

انطلاقًا من هذا يمكننا أن نعطي غاية خاصة لكل برمجيَّة، فمثلًا معالج النصوص موجود لمساعدة الناس على الكتابة، والمتصفح موجود لمساعدتهم على تصفح الشبكة العنكبوتيَّة.

بعض البرمجيّات تساعد مجموعة محددة من الناس، فمثلًا هناك العديد من برمجيَّات المحاسبة أُنشِئَت لمساعدة المحاسبين لا غير، فهي تستهدفهم وحدهم فقط.

وحتى البرمجيات التي تساعد الحيوانات والنباتات، فغايتها الحقيقية هي مساعدة البشر على مساعدة الحيوانات والنباتات.

المهم أنَّ البرمجيَّات لا تساعد الجمادات، فالبرمجيَّات موجودة دائمًا لمساعدة الناس وليس الحاسوب. فحتى عندما تكتب مكتبات برمجيَّة فأنت تكتبها لمساعدة المبرمجين الذين هم بدورهم بشر وليس لمساعدة الحاسوب.

2. ليس هناك كلمة مناسبة لوصف هذا النوع من الحقائق (الغاية من البرمجيات). لا يمكننا تسميّة هذه الحقيقة بالقانون، فهي لا تتميز بخصائصه، فمثلًا هذه الحقيقة لا تتنبأ بالمستقبل. ومع ذلك فهي أهم من أي قانون. على كلٍ، سندرجها في قائمة القوانين في الملحق آخر الكتاب، وسنُشير إليها فيما تبقى بعبارة "غاية تصميم البرمجيات".

فما الذي تعنيه كلمة "مساعدة"؟ معنى الكلمة شخصي نوعًا ما، فالذي يساعد شخصًا قد لا يساعد غيره، لكن للكلمة تعريف مُحدَّد في القاموس فالأمر ليس متروكًا لكل شخصٍ ليدلي بدلوه. يُعرِّف قاموس المعاني الجامع كلمة مساعدة كالتالي:

مصدر من ساعد بمعنى إعانة أو معونة، وساعده: عاونه أي قدم له معونة.

هناك العديد من الطرق للمساعدة، مثل ترتيب المواعيد أو كتابة كتاب أو التخطيط لحمية أو أي شيء آخر. لك الحرية في اختيار بما تساعد، ولكن الغاية دائمًا أن تساعد.

ليست غاية البرمجيَّات "جني النقود" أو "استعراض ذكائك"، وأي شخص يُبَرمِج بهذه الغايات سوف يتعارض مع الغاية الحقيقية للبرمجيَّات، وبالتالي من المُحتمَل أن يقع في المشاكل. بالطبع تلك الطرق أيضًا "لمساعدة" نفسك، ولكن مجال هذا النوع من المساعدة ضيّق، والتصميم لغايات كهذه هذه سيؤدي غالبًا إلى برمجيَّات أدنى جودة من تلك المُصمَّمة خصيصًا لمساعدة الناس على القيام بما يرغبون أو يحتاجون فعله[3].

3. لا مُشكلة في أن يكون "جني النقود" غايتك الشخصية أو غاية منظمتك، فقط لا يجب أن يكون غاية برمجيّتك. ترتبط في جميع الحالات كميّة جنيك للنقود مباشرةً مع مقدار المساعدة التي تقدمها برمجيّتك للناس. الحقيقة أنَّ العاملان الأساسيان اللذان يحددان دخل شركتك البرمجيّة هما المهارات الاقتصادية (من إدارة وتسيير وتسويق ومبيعات) ومقدار مساعدتك للناس.

المبرمجون الذين لا يفكرون في مساعد الناس برمجياتهم سيئة لأنَّها تساعد أقل. يُمكِن تفسير الأمر على النحو التالي (هذا رأي مبني على ملاحظة العديد من المبرمجين عبر سنين تجربتي): قدرتك المحتملة على كتابة برمجيات جيدة محدودة بقدرتك على مساعدة الآخرين.

باختصار، عند اتخاذنا لقرارات متعلقة بالبرمجيَّات، فالسؤال الذي نسأله دائمًا هو: "كيف يمكن أن نساعد؟" (وتذكر أنَّه هناك درجات مختلفة من المساعدة، فهنالك ما يساعد بشكل كبير أو صغير وما سيساعد الكثير من الناس أو قلة منهم). هذا يستطيع إعانتك حتى على تحديد الميزات ذات الأولويَّة، فالميزة التي ستساعد الناس كثيرًا ستُعطى الأولويَّة القصوى. هناك المزيد لتعرفه عن تحديد أولويَّة الميزات، ولكن ما يهم أن تعرفه الآن أنَّ السؤال عن مدى مساعدة ميّزة ما للمستخدمين طريقة جيدة وأساسيَّة لاتخاذ القرارات بشأن الميّزات المقترح إضافتها لنظامك البرمجي.

مساعدة الناس عامةً هي أهم شيء عليك وضعه في حسبانك عند تصميمك للبرمجيات، فَتعريفُها ما مكَّننا الآن من إنشاء وفهم بطريقة صحيحة علم تصميم البرمجيات.

تطبيق واقعي

كيف نُطبِّق الغاية من البرمجيات على مشاريعنا في أرض الواقع؟ لنفترض أنّنا نُبرمِج محرر نصوص للمبرمجين. أول شيء نحتاج تحديده هو الغاية من البرنامج. فلنبقي الأمر بسيطًا، ولنعتبر أنَّ الغاية هي "مساعدة المبرمجين على تحرير النصوص". لا ضرر في المزيد من التفاصيل بل أحيانًا تكون مفيدة، لكن إن لم يستطع الفريق الاتفاق على غاية محددة فمن الأحسن أن نأتي بغاية بسيطة على الأقل كالسابقة.

الغاية في أيدينا، فلنلقي الآن نظرة على الميزات المقترحة. سنُخضِع كل واحدة للتساؤل التالي: "كيف ستُساعِد هذه الميزة المبرمجين على تحرير النصوص؟". إن كان الجواب أنَّها "لن تساعدهم " سنشطبها فورًا من القائمة. سنجيب بعدها بعبارة صغيرة عن السؤال في الميزات المتبقيّة. مثلًا إن طُلب مِنَّا إضافة اختصارات لوحة المفاتيح للأوامر المشهورة، لأجبنا: "هذا سيساعد المبرمجين على تحرير النصوص لأنَّه يُوفر عليهم عناء التوقف عن الكتابة" (لا داعي لتدوين الإجابة، معرفتها ووضعها في ذهنك يكفي).

هناك عدّة أسباب مُفيدة أُخرى لسؤال هذا السؤال:

- سيُساعدك على التثبُّت من الميزة ومعرفة كيف ستُنفَّذ، فمثلًا من الإجابة السابقة عن الاختصارات نعلم أنَّه يجب أن نُنفِّذ الميزة فورًا؛ وذلك للقيمة التي ستعود على المبرمجين منها.
- سيُساعِد فريقك على الخروج باتفاق حول قيمة الميزة. البعض لن تعجبه فكرة الاختصارات، لكن الجميع سيتفق أنَّ الشرح السالف يوضح قيمتها. لربما حتى بعض المبرمجين سيأتون بفكرة أخرى تلبي حاجة المستخدمين (التعامل مع المحرر بسرعة أكبر) دون الاختصارات، وهذا جيد إذا أدت الفكرة لميزة جديدة أفضل لتنفيذها بدلًا من القديمة. الإجابة ستدلنا على اللازم حقًا وليس على ما يعتقد المُستخدِم أنَّه يحتاج.
- ترتيب الميزات على حسب أولويتها، وهذا سيساعد مدير المشروع على تنظيم العمل.
- حذف الميزات غير المهمة عندما تسوء الأمور ويمتلأ البرنامج بالميزات.

يمكننا كذلك أن نضع قائمة بالعلل، حيث نسأل السؤال المعاكس: " كيف تُعيق هذه العلّة المبرمجين من تحرير النصوص؟". أحيانًا الإجابة واضحة، فلا حاجة مثلًا لشرح مساوئ انهيار البرنامج عند محاولة حفظ الملف. هناك الكثير من الطرق الأخرى لتطبيق الغاية من البرمجيات في عملك اليومي، وهذه ما كانت سوى أمثلة بسيطة.

أهداف تصميم البرمجيّات

والآن، وقد عرفنا الغاية من البرمجيّات، أصبح بإمكاننا التوجُّه قليلًا للتكلُّم عن علم تصميم البرمجيّات. تعلَّمنا أثناء الحديث عن غاية البرمجيَّات، أنَّنا عندما عندما نكتُب برمجيَّة نحن نحاول مساعدة الناس. وبالتالي إحدى أهداف علم تصميم البرمجيّات ينبغي أن تكون:

تمكيننا من كتابة برمجيّات مفيدة قدر الإمكان.

ثانيًا، عادةً ما نرغَب في الاستمرار بمساعدة الناس ببرمجيّاتِنا، وبالتالي الهدف الثاني من علم تصميم البرمجيّات هو:

تمكين برمجياتنا من الاستمرار في أن تكون مفيدة قدر الإمكان.

هذا هدف رائِع على ما يبدو، ولكن أي نظام برمجيّ بأي حجم سيكون غاية في التعقيد، ما يجعل الاستمرار في جعله مفيدًا مع مرور الزمن مهمة ليست بالهيّنة. العائِق الرئيسي الذي يواجهنا هذه الأيام في كتابة وصيانة البرمجيَّات المفيدة هو صعوبة التصميم والبرمجة. عندما تكون البرمجيَّات صعبة الإنشاء أو التعديل، يقضي المبرمجون معظم وقتهم تركيزهم مصبوب على جعل الأمور تعمل وحسب، والباقي من وقتهم، القليل بالنسبة للوقت الكلي، على مساعدة المُستخدِم. بينما إن كان من السهل العمل على النظام البرمجيّ، سيقضي المُبرمجون وقتًا أكبر في التركيز على جعل البرمجيَّة مُفيدة للمستخدمين ووقتًا أقل في التركيز على تفاصيل البرمجة. بنفس المقياس، كلما كان أسهل صيانة أجزاء البرمجيَّة، كلما كان أسهل علينا كمبرمجين الحرص على الحفاظ على استمراريَّة كون البرمجيَّة مُفيدة.

كُل ذلك يقودنا للهدف الثالث من تصميم البرمجيّات، والذي هو:

تصميم برمجيَّات يُمكِن إنشاؤها وصيانتها بيُسِر وسهولة قدر الإمكان بواسطة مُبرمجيها، وبالتالي يُمكن أن تكون - وأن تستمر في أن تكون - مُفيدة قدر الإمكان.

يُعتقَد تقليديًّا أنَّ هذا الهدف هو الهدف من تصميم البرمجيّات، وبالرغم من أنَّ ذلك غير مذكور صراحةً. من المهم كذلك امتلاك الهدف الأوَّل والثاني، جنبًا إلى جنب مع الهدف الثالث، لإرشادنا. حيث أنَّنا نريد أن نتذكَّر أنَّ كون برمجياتنا مُفيدة الآن وفي المستقبل هي الدوافع وراء هذا الهدف الثالث.

شيء آخر من المهم الإشارة إليه حيال هذا الهدف هو عبارة "بيُسِر وسهولة قدر الإمكان". المغزى من الفكرة هنا جعل برامجنا سهلة الإنشاء والصيانة وليس جعلها صعبة أو معقَّدة. لا يعني ذلك أنَّ كل شيء ينبغي أن يكون بسهولة فوريَّة، فقد يأخذ الأمر بعض الوقت لتعلُّم تقنية جديدة أو لتصميم شيء حَسَن البنية، ولكن الفكرة أنَّ خياراتك، على المدى البعيد، ينبغي أن تجعل إنشاء وصيانة برمجيّاتك أسهل.

يتداخل أحيانًا الهدف الأوَّل (كون البرمجيَّة مفيدة) مع الهدف الثالث (سهولة الصيانة) قليلًا، حيث أنَّ جعل برمجيّاتك مُفيدة قد يؤدي إلى جعلها صعبة الصيانة في نفس الوقت. كانا هذان الهدفان تاريخيًّا متداخلان إلى حدٍ كبير أكثر من اللازم، فمن الممكن قطعًا كتابة أنظمة برمجيَّة تُصان وبالغة الإفادة للمُستخدمين بنفس الوقت. وحقيقةً، إن لم تجعل برمجيّتك قابلة للصيانة، سيكون من الصعب جدًا موافاة الهدف الثاني في الاستمرار في جعلها مُفيدة. ما يجعلنا نستخلص أنَّ الهدف الثالث مهم جدًا؛ كونه لا يمكن تحقيق الهدفين الآخرين من دونه.

الفصل الرابع: المستقبل

أوَّل سؤال يواجه مصممي البرمجيات هو: "كيف أتخذ قرارات بشأن البرمجيَّات التي أصممها؟". عندما تواجهك العديد من الخيارات المتاحة، أيٌّ من هذه الخيارات سيكون الأفضل؟ إنَّها ليست مسألة أيّ هذه القرارت هو الخيار **الصحيح تمامًا** وأيّها **خاطئ تمامًا**. بل ما يجب أن نعرفه هو: "مِنْ هذه القرارات المحتملة، أيّها أفضل من الأخرى؟". إنَّها مسألة تقييم هذه القرارات، ثم اختيار أفضل قرار من بين كل القرارات المحتملة. فمثلًا، ربما قد يسأل مصمم نفسه: "لدينا 100 ميزة مختلفة يمكننا أن نعمل عليها اليوم، لكن لدينا الطاقة التي تُمكِّنُنا من العمل على ميزتين منها فقط. بأيّ هذه الميزات علينا أن نبدأ؟".

معادلة تصميم البرمجيات

السؤال السابق وأي سؤال آخر من هذا النوع في مجال تصميم البرمجيات يمكن أن تجيب عليه هذه المعادلة:

$$ر = \frac{ق}{ج}$$

حيث أنَّ:

- ر يُمثِّل مقدار رغبة القيام بالتغيير.
- ق يُمثِّل مقدار القيمة الراجعة من القيام بالتغيير. عادةً يمكنك أن تجدها عبر السؤال: "ما مدى المساعدة التي سيقدمها التغيير الجديد إلى المستخدم؟"، وتوجد بعض الطرق الأخرى كذلك.
- ج يُمثِّل مقدار الجهد المبذول في سبيل هذا التغيير.

ما تخبرنا به هذه المعادلة:

تتناسب الرغبة بإجراء تغيير طردًا مع القيمة العائدة مِنه وعكسيًّا مع مقدار الجهد المبذول لإجراءه.

هذه المعادلة لا تخبرك ما إذا كان هذا التغيير صحيح أو خاطئ **بالمطلق**، بل تخبرك كيف **تقيِّم** خياراتك. التغييرات التي سترجع بقيمة كبيرة وتتطلَّب جهد قليل "أفضل" من تلك التغييرات التي سترجع بقيمة قليلة وتتطلب جهد كبير.

حتى إذا كان سؤالك: "هل يجب أن نبقى على حالنا **دون** إحداث تغيير؟" فستُجيب هذه المعادلة على سؤالك. اِسأل نفسك عندها: "ما هي القيمة العائدة من البقاء على الوضع الحالي؟" و "ما مقدار الجهد المطلوب للبقاء على الوضع الحالي؟" وقارنهما مع القيمة العائدة من التغيير والجهد المبذول في إحداثه.

القيمة

ما الذي نقصده بـ"القيمة" في هذه المعادلة؟ أبسط تعريف للقيمة سيكون:

الدرجة التي يساعد بها هذا التغيير أي شخص في أي مكان.

أهمُّ الناسِ المستهدفين بالمساعدة هم مستخدمو برمجيّاتك. مع ذلك، إضافة الميزات التي ستفيدك ماليًا هي أيضًا شكل من أشكال القيمة، فهي قيّمة بالنسبة لك. هناك عدة أشكال حقيقةً يمكن أن يكون التغيير بها له قيمة، و ما سبق مُجرَّد مثالان.

بعض الأحيان يكون إيجاد القيمة الحقيقية وبأرقام دقيقة صعبًا. فمثلًا لنقل أنَّ برمجيّتك تساعد المستخدمين على خسارة الوزن، كيف ستحسب مقدار مساعدة شخص ما على خسارة وزنه بالضبط؟ لن تتمكن من ذلك. لكنك **تستطيع** أن تعرف بدقَّة أنَّ بعض ميِّزات هذه البرمجيَّة تساعد المستخدمين على

خسارة أوزانهم **كثيرًا** وأنَّ بعض الميزات الأخرى لن تساعد على ذلك **أبدًا**. لذا مازال بإمكانك تقييم التغييرات بقِيَمِها.

إنّ فهم قيمة كل تغيير مُحتمَل غالبًا ما يأتي من الخبرة كمطوّر ومن إجراء أبحاث مناسبة مع المستخدمين لإيجاد ما الذي سيساعدهم أكثر.

احتماليَّة وأهميّة القيمة

القيمة مكوَّنة من عاملين: **احتماليَّة القيمة** (ما احتمالية أن يستفيد المستخدم من التغيير)، **وأهميَّة القيمة** (ما مدى استفادة المستخدم من هذا التغيير).

على سبيل المثال:

- الميزة التي قد تنقذ حياة شخص ما، حتى لو كانت احتمالية الحاجة إليها واحد من مليون، هي ميزة قيّمة. هذه الميزة لها أهميَّة كبيرة (إنقاذ حياة)، على الرغم من أن لها احتمالية ضئيلة أن تكون قيّمة.

  مثال آخر: ربما تضيف على برنامج جداول ميزة تساعد الكفيفين على إدخال الأرقام إلى النظام. نسبةٌ صغيرة من الناس كفيفون، ولكن بدون هذه الميزة لن يتمكّنوا من استخدام برمجيّتك **أبدًا**. مرّة أخرى، هذه الميزة قيّمة لأنها مهمَّة، على الرغم من أنها تؤثّر على نسبة ضئيلة من المستخدمين (احتماليَّة القيمة ضئيلة).

- إذا كانت هناك ميزة تجعل 100٪ من مُستخدميك يبتسمون فهي ميزة قيّمة أيضًا. هي ميزة ذات أهمية صغيرة (جعل الناس يبتسمون)، لكنها تؤثر على عدد كبير من المستخدمين، فاحتماليَّة قيمتها عالية هنا.

- إذا أضفت ميزة ذات فرصة ضئيلة في جعل الشخص يبتسم، فهذه ليست بالميزة القيمة. فكل من أهمية هذه الميزة واحتماليَّة قيمتها ضئيتلان.

لذلك، عند إضافة الميزات، يجب النظر في قيمتها، حيث يجب مراعاة ما يلي:

- كم من المستخدمين (النسبة المئوية) الذين سيستفيدون من هذا التغيير (سيكون قيّم بالنسبة لهم)؟
- ما هي احتمالية استفادة المستخدم من هذا التغيير؟
- متى تكون هذه الميزة مهمة؟ وكم ستكون أهميتها (أي نفعها)؟

توازن الضرر

بعض التغييرات قد تُحدِث ضررًا إلى جانب النفع الذي تجلبه. فمثلًا بعض المستخدمين قد ينزعجون من الإعلانات التي يعرضها برنامجك، حتى إذا كانت هذه الإعلانات تساعدك كمطور. عند حساب قيمة الميزة يجب النظر في مقدار الضرر الذي ستلحقه و موازنته مع النفع الذي ستجلبه.

قيمة امتلاك مُستخدمين

ليس للميزات التي ليس لها مستخدمون قيمة فورية. هذه الأخيرة تتضمن الميزات التي لا يجدها المستخدمون أو الميزات الصعب استخدامها أو الميزات التي لا تساعد أحدًا. ربما سيكون لهذه الميزات قيمة في المستقبل، لكن الآن لا قيمة لها.

هذا يعني أنّه في معظم الحالات يجب عليك إطلاق برنامجك بغاية أن يكون نافعًا ذو قيمة. التغيير الذي تطول إضافته قد يُصبِح بلا قيمة؛ لأنَّه لا يُضاف في الوقت المناسب أي الوقت الذي يساعد الناس بفعالية. قد يكون من المهم أخذ مواعيد الإصدار في الحسبان عند تحديد رغبة التغييرات.

الجهد

تمثيل الجهد بالأعداد أسهل منه في تمثيل القيمة. يمكن وصف الجهد عادةً بأنَّه عدد معين من ساعات العمل من عدد معين من الأفراد. فمثلًا: "100 شخص/سنة" مقياس عددي شائِع الاستخدام للجهد، ويُمثِّل 100 سنة من العمل من شخص واحد، سنة كاملة من العمل من 100 شخص، سنتين من العمل من 50 شخص... إلخ.

على الرغم من أنّه يمكننا تمثيل الجهد بواسطة الأعداد، إلا أنَّ قياسه عمليًّا يمكن أن يكون صعبًا وربما حتى مستحيلًا. التغييرات يمكن أن تخفي الكثير من التكاليف والتي يكون من الصعب توقُّعها، كالوقت الذي ستقضيه مستقبلًا في إصلاح أي علَّة نتَجَت عن تغييرٍ ما. ولكن إن كنت خبيرًا في مجال تطوير البرمجيات، فسيمكنك أن تُرتِّب هذه التغييرات من خلال معرفة كميَّة الجهد المحتملة التي ستتطلبها حتى وإن لم تكن تعلم العدد الدقيق لكلٍ منها.

عندما ننظر إلى الجهد الذي يتضمنه التغيير فإنَّه من المهم أن نأخذ بعين الاعتبار حساب كل الجهود الممكن أن تدخل، ولا نهتم فقط بالوقت الذي سنستغرقه في البرمجة. ما هو عدد البحوث التي سيتطلبها هذا التغيير؟ ما هو عدد الاتصالات التي سُيجريها المطورون فيما بينهم؟ ما هو الوقت الذي ستستغرقه في التفكير في التغيير؟

مختصر القول أنَّ كل جزء من الوقت مرتبط بالتغيير هو جزء من تكلفة الجهد.

الصيانة

المعادلة التي نمتلكها لحد الآن جد بسيطة، ولكنها تفتقر لعامل مهم ألا وهو الوقت. ليس عليك فقط أن تُنفِّذ التغيير ولكن عليك صيانته مستقبلًا ومع مرور الوقت. فكل التغييرات تتطلب الصيانة وهذا الأمر بديهي مع بعضها. لو كنت مثلًا تكتب برنامجًا يحسب القيمة الضريبيَّة للأشخاص سيكون عليك تحديثه كل عام فيما يتوافق مع القانون الضريبي الجديد. وحتّى التغييرات التي لا يبدو فورًا أنَّها ستكون ذات تكلفة صيانة على المدى البعيد، فإنَّها ستُكلِّفك، حتى وإن كانت فقط تكلفة التأكد من أنَّ الكود مازال يعمل عندما ستختبره العام المقبل.

علينا كذلك أن نأخذ بعين الاعتبار القيم لكل من الفترة الآنية والمستقبلية. فعندما نُنفِّذ بعض التغييرات على أنظمتنا هذا سيساعد مستخدمينا الحاليين وقد يساعد كذلك مستخدمينا المستقبليين، بل ومُمكِن حتى أن يؤثر على عددهم الكلي مستقبلًا، وبالتالي سيُغيّر القدر الذي تساعد فيه برمجيتنا ككل الناس.

تتغيّر حتى قيمة بعض الميزات بمرور الوقت، فمثلًا امتلاكك لبرنامج ضريبي يفهم القانون الضريبي لعام 2009م سيكون قيّمًا ما بين العامين 2009م-2010م ولكن بمجرد حلول العام 2011م فإنه يفقد قيمته، وبالتالي تفقد هذه الميزة قيمتها مع مرور الوقت. هناك ميزات كذلك، على عكس ما رأيناه سابقًا، تزداد قيمتها بمرور الزمن.

من خلال النظر إلى هذا بواقعيَّة، نرى بأنّ الجهد الفعلي يشتمل على كل من الجهد التنفيذي وجهد الصيانة، و كذلك الحال بالنسبة للقيمة، فهي تشتمل على القيمة الآنية والقيمة المستقبليّة. بصياغة ذلك رياضيًّا نحصل على:

$$ج = ج_{ت} + ج_{ص}$$

$$ق = ق_{آ} + ق_{م}$$

حيث أنَّ:

- $ج_{ت}$ يُمثِّل الجهد التنفيذي.
- $ج_{ص}$ يُمثِّل جهد الصيانة.
- $ج_{آ}$ يُمثِّل القيمة الآنيّة.
- $ج_{م}$ يُمثِّل القيمة المستقبليّة.

المعادلة الإجمالية

عند وضع جميع العناصر السابقة مع بعض، تصبح المعادلة الإجمالية على الشكل التالي:

$$ر = \frac{ق_{آ} + ق_{م}}{ج_{ت} + ج_{ص}}$$

عند ترجمتها لغويًّا تصبح:

تتناسب الرغبة في التغيير طردًا مع مجموع كل من القيمة الآنيّة والقيمة المستقبلية مقسومةً على مجموع كل من الجهد التنفيذي وجهد الصيانة.

هذه الأخيرة تُمثِّل المعادلة الأساسيّة في تصميم البرمجيات، إلا أنّه لا زالت هنالك بعض الحيثيّات لتعلمها حول هذه المعادلة.

اختزال المعادلة

كل من "القيمة المستقبلية" و"جهد الصيانة" مُتعلقين بالزمن، والذي يُؤدي إلى إحداث أمور هامة حول المعادلة عند تطبيقها على مواقف واقعيّة. لتوضيح الفكرة، دعنا نفترض بأنَّه يمكننا استعمال المال لتمثيل كل من القيمة والجهد لحل هذه المعادلة. وعليه: سُتحسَب "القيمة" بكميّة المال التي سيجلبها التغيير، بينما "الجهد" فسيُحسَب تبعًا لكميّة المال التي ستكلفنا لتنفيذ التغيير. ليس عليك أن تستعمل هذه المعادلة بنفس هذه الكيفيَّة على أرض الواقع، ولكننا قمنا بهذا التمثيل من أجل التبسيط وحسب.

دعنا نفترض أنَّنا نرغب القيام بتغيير لمعادلته هذا الشكل:

$$\text{ر} = \frac{\$\ 10,000 + \$\ 1000/\text{يوم}}{\$\ 1,000 + \$\ 100/\text{يوم}}$$

بمعنى آخر يُفسِّر المعادلة أعلاه، هذا التغيير يُكلِّف 1,000$ لتنفيذه (الجهد التنفيذي الكائِن في المقام من الجهة اليسرى) مما يُكسِبُنا 10,000$ مباشرةً (القيمة الآنيّة الكائِنة في البسط من الجهة اليسرى). بعد كل يوم من ذلك اليوم (الذي نكسَب فيه القيمة الآنيّة)، سيُكسبنا التغيير 1,000$ (القيمة المستقبليّة الكائِنة في البسط من الجهة اليمنى) وسيُكلفنا 100$ لصيانته (جهة الصيانة الكائِن في المقام من الجهة اليمنى).

ستبلغ القيمة المستقبليّة الكليّة، بعد مرور عشرة أيام على التنفيذ، القيمة 10,000$ وجهد الصيانة الكلي 1,000$. تساوي القيمة المستقبليّة تلك نفس "القيمة الآنيّة" الأصليّة وتساوي تكلفة جهد الصيانة نفس تكلفة التنفيذ الأصليّة، وكل ذلك في عشرة أيام وحسب.

ستبلغ القيمة المستقبليّة الكليّة بعد مئة يوم 100,000$ وجهد الصيانة الكلي 10,000$.

ستبلغ القيمة المستقبليّة الكليّة بعد ألف يوم 1,000,000$ وجهد الصيانة الكلي 100,00$. في النقطة التي تُصبِح فيها القيمة المستقبليّة كذلك، ستصبح "القيمة الآنية" الأصليّة وتكلفة التنفيذ غاية في الصغر في المقارنة، ومع مرور الوقت ستُصبِح أكثر صغرًا حتى تختفي أهميتها بالكامل نهايةً. وبالتالي، ومع مرور الوقت، ستُختزَل معادلتنا لتصبِح[4]:

$$ر = \frac{ق_{م}}{ج_{ص}}$$

معظم القرارات حقيقةً في تصميم البرمجيّات تُختزَل بالكامل لتقتصر على قياس القيمة المستقبليّة للتغيير على جهد صيانته. توجد بعض الحالات التي يكون فيها جهد التنفيذ الأصلي كبيرًا كفاية ليكون ذا أهمية في القرار، ولكن حالات كهذه قليلة الحدوث. تكون عامةً البرمجيَّة مُصانة طالما ضُمِنَ أنَّ القيمة الآنيّة والجهد التنفيذي سيُصبِح عديم الأهميّة في جميع الحالات تقريبًا مقارنةً مع القيمة المستقبليّة وجهد الصيانة على المدى البعيد.

4. ملاحظة اختياريّة للرياضيين: إذا كنت قد درست التفاضل والتكامل من قبل، قد تكون لاحظتَ أنَّنا بدأنا في تحليل حد المعادلة مع اقتراب الوقت من اللانهاية. ينبغي أن يُنظَر عامةً لمعادلة تصميم البرمجيّات على أنَّها سلسلة لانهائيَّة ذات حد وليست مجرَّد معادلة ساكنة، إلا أنَّنا كتبناها هنا كمعادلة ساكنة للتبسيط.

ما ترغب وما لا ترغب بحدوثه

الدرس الرئيسي لتعلمه هنا أننا يجب أن نحاول تجنُّب المواقف التي يفوُق فيها جهد الصيانة لتغيير قيمته المستقبليّة. لنفترض مثلًا أنَّنا أجرينا تغيير جهد صيانته وقيمته المستقبليّة على مدار خمس أيام موضَّحة في الجدول أدناه:

اليوم	الجهد	القيمة
الأوَّل	$10	$1,000
الثاني	$100	$100
الثالث	$1,000	$10
الرابع	$10,000	$1
الخامس	$100,000	$0.10
المجموع الكلِّي	$111,110	$1111.10

من الواضِح من الجدول أنَّ ذلك التغيير شديد السوء ولا ينبغي علينا إجراء مثله. إذا استمرَّ جهد الصيانة والقيمة بهذا المُعدَّل لن تكون قادرًا على الاستمرار في صيانة النظام؛ وذلك لكون جهد الصيانة مُكلِّف للغاية مقارنةً بالقيمة التي تكسبها والتي ستصبح صفرًا.

أي موقف يزداد فيه جهد الصيانة بمعدَّل أسرع من معدل ازدياد القيمة سيوقعك في المشاكل وإن بدا المُعدَّل بدايةً مقبولًا:

اليوم	الجهد	القيمة
الأوَّل	$1000	$1000
الثاني	$2000	$2000
الثالث	$4000	$3000

الرابع	$8000	$4000
المجموع الكلِّي	$15,000	$10,000

الحل المثالي، والطريقة الوحيدة لضمان النجاح، هو أنَّ تُصمِّم أنظمتك البرمجيّة بطريقة يتناقص فيها جهد صيانتها مع مرور الوقت، ليصل نهايةً إلى الصفر (أو إلى قيمة مقاربة قدر الإمكان من الصفر). طالما كان بإمكانك فعل ذلك فلا حاجة لتقلق بشأن مقدار كِبَر أو صِغَر القيمة المستقبليّة الذي ستُصبِح عليه. يوضِّح الجدولان أدناه بعض الحالات المرغوبة من هذا النمط:

اليوم	**الجهد**	**القيمة**
الأوَّل	$1,000	$0
الثاني	$100	$10
الثالث	$10	$100
الرابع	$0	$1,000
الخامس	$0	$10,000
المجموع الكلِّي	$1,110	$11,110

اليوم	**الجهد**	**القيمة**
الأوَّل	$20	$10
الثاني	$10	$10
الثالث	$5	$10
الرابع	$1	$10
الخامس	$0	$10
المجموع الكلِّي	$36	$50

لا تزال التغييرات ذات القيمة المستقبليّة الأعلى أكثر استحسانًا، ولكن طالما كان لكل قرار تكلفة صيانة تقترب إلى الصفر مع مرور الوقت، لن تَقَع في موقف خطير مستقبلًا.

طالما كانت القيمة المستقبليّة، نظريًّا، أكبر دومًا من تكلفة جهد الصيانة، طالما كان التغيير مرغوبًا. وبالتالي، يُمكنك إجراء تغييرات تزداد فيها القيمة المُستقبليّة وتكلفة جهد الصيانة معًا، وذلك طالما كانت القيمة المستقبليّة كبيرة كفايةً لتفوق تكلفة جهد الصيانة:

اليوم	الجهد	القيمة
الأوّل	$1	$0
الثاني	$2	$2
الثالث	$3	$4
الرابع	$4	$6
الخامس	$5	$8
المجموع الكلّي	$15	$20

تغيير كهذا ليس بذلك السوء، ولكن من المرغوب أكثر إجراء تغييرات تتناقص تكلفة صيانتها مستقبلًا، حتى لو كان جهد تنفيذها كبيرًا. تزداد الرغبة بالتغيير، إذا ما كان جهد صيانته في تناقُص، مع مرور الزمن، ما يجعله الاختيار الأفضل مقارنةً مع باقي الاحتمالات.

غالبًا ما يتطلّب إنشاء نظام جهد صيانته سيتناقص مُستقبلًا جُهدًا تنفيذيًّا بالِغ الكِبَر (أي المزيد من أعمال التخطيط والتصميم). بالرغم من التزايد في الجهد التنفيذي ذلك، إلا أنّه عليك أن تتذكر أنّ الجهد التنفيذي تقريبًا دائمًا ما يكون بعامل غير مُهم في اتخاذ القرارات التصميميّة، ويبنغي عمومًا تجاهله.

مُختصَر القول:

تخفيض جهد الصيانة أكثر أهميّة من تخفيض الجهد التنفيذي.

هذه القاعدة من أهم الأشياء التي عليها معرفتها عن تصميم البرمجيّات.

غالبًا أنت تتساءل الآن عن الذي يُسبِّب جهد الصيانة ذاك، وعن الكيفيَّة التي يُمكننا عبرها بناء نظام جهد صيانته يتناقص بمرور الوقت. سنتكلَّم عن ذلك الموضوع في غالبيّة ما بقي من كتابنا، ولكننا سنُعرِّج قبل ذلك للتكلُّم قليلًا بعد عن المُستقبَل.

جودة التصميم

من السهل جدًّا كتابة برمجيّة تُساعد شخصًا واحدًا حالًا، ولكن من الصعب جدًّا كتابة واحدة تُساعد ملايين المُستخدمين وتستمر بفعل ذلك لعقود مُستقبلًا. ولكن أين ستُكون مُعظم الجهود البرمجيّة، ومتى ستستخدم الغالبيّة العظمى من هؤلاء المُستخدمين البرمجيّة؟ حالًا أم في العقود القادمة تلك؟

الجواب هو أنَّ العمل البرمجي والمُستخدمون سيتزايدون في المُستقبَل أكثر من الحاضر. على برمجيّتك أن تُنافِس وتتواجَد في المُستقبَل، وجهد الصيانة وعدد المُستخدمين سيتزايد.

عند تجاهل حقيقة أنَّ هناك "مُستقبلًا" والاستمرار في صنع أشياء "تعمل وحسب" في الحاضِر، ستُصبِح برمجياتنا صعبة الصيانة. كون برمجياتنا صعبة الصيانة سيجعل من الصعب الاستمرار في إبقائها مُفيدةً للناس (والذي هو أحد أهداف تصميم البرمجيَّات). إذا لم يكن بإمكانك إضافة ميزات جديدة وإصلاح المشاكل السابقة، ستحصل نهايةً على "برمجيّة سيئة" غير مُفيدة ومليئة بالعلل.

هذا يقودنا إلى القاعدة التالية:

> **ينبغي أن تكون جودة تصميمك متناسبة مع المدى الزمني المُستقبلي الذي سيستمر نظامك في مساعدة الناس خلاله.**

فإذا كنت، مثلًا، تسعى لتصميم برمجيَّة ستعمل لبضعة ساعات فقط، لن تضطر لبذل جهد كبير في تصميمها. بينما إن كانت برمجيّتك ستُستخدم لعشرة سنوات مقبلة (والذي يحدُث أكثر مما تتوقَّع، حتى وإن كنت تظن بدايةً أنَّها ستُستخدَم فقط لست شهور أو ما شابه)، فعندها ستحتاج لبذل جهد كبير في تصميمها. عندما تشك في المدى الزمني المُستقبلي لبرمجيّتك، صمِّمها كما لو أنَّها ستُستخدَم للأزل: لا

تُقيّد نفسك بطريقة مُعينة للقيام بالأشياء، وابقِ الأمور مَرِنة قدر الإمكان، وحاول أن لا تتخذ أي قرار لن تستطيع تغييره، وركّز جيدًا على تصميم البرمجيّة.

عواقِب غير متوقعة

إذًا علينا، أثناء تصميم البرمجيّات، وضع المُستقبَل نِصاب أعيننا كهدفٍ أساسي. بالرغم من أنَّ المُستقبَل شيءٌ مهم، إلا أنَّه هناك شيء من أهم الأشياء حول أي نوع من الهندسات عليك معرفته:

هناك بعض الأشياء لا يمكنك التنبؤ بها حول المُستقبَل.

في مجال تصميم البرمجيّات حقيقةً، لا يمكنك تنبؤ معظم الأشياء حول المُستقبَل.

إحدى أكثر الأخطاء التي يَقع فيها المبرمجون كارثيّةً وشيوعًا هي محاولة التنبؤ بشيء مُستقبلي متجاهلين حقيقة أنَّهم لا يمكنهم ذلك.

فلنتصوَّر، كمثال، أنَّ مُبرمِجًا كتب برمجيَّة عام 1985م تُصلِح الأقراص المرنة المُعطلة. لا يُمكِن لهذه البرمجيّة إصلاح أي شيء عدا الأقراص المرنة، فَكُلُ جزءٍ منها مُعتمِد على الكيفيَّة التي تعمل بها هذه الأقراص. برمجيَّة كهذه بحلول الآن ستكون منسيّة بالكامل؛ وذلك لأنَّ لا أحد هذه الأيام يستخدم الأقراص المرنة. مُبَرمجِ هذه البرمجيَّة تنبَّأ "أنَّ الناس سيستمرون باستخدام الأقراص المرنة للأبد" وهو شيء لا يمكنه معرفته حقًا.

قد يكون من المُستطاع التنبؤ بالمُستقبَل القريب، ولكنَّ المُستقبل البعيد غير معروف بتاتًا. المستقبل البعيد كذلك أكثر أهميّة لنا من القريب؛ وذلك لأنَّ قراراتنا سيكون لها عواقب أكثر في ذلك المدى.

> أنت أأمَن إذا لم تحاول التنبؤ بالمُستقبل بتاتًا. بل اتخاذ، عوضًا عن ذلك، قراراتك بناءً على المعلومات المباشرة المعروفة في الزمن الحاضر.

قد يبدو هذا الكلام مُتضاربًا مع ما كُنَّا نقوله من بداية الفصل، ولكنَّه ليس كذلك. المستقبل هو أهم شيء لوضعه في عين الاعتبار أثناء اتخاذ القرارات، ولكن هناك فرق بين التصميم بطريقة تسمح بتغييرات مستقبليّة ومحاولة التنبؤ بالمستقبل.

كمثال دعنا نَقُل أنَّك مُخيَّر بين الأكل والموت جوعًا. أنت لست بحاجة للتنبؤ بالمُستقبل لاتخاذ قرارك هنا، فأنت بالتأكيد تعرف أنَّ الأكل هو الخيار الأفضل. كيف تعرف؟ لأنَّ الأكل ببساطة سيبقيك حيًّا الآن، والبقاء حيًّا أفضل من الموت. المُستقبَل مهم، وعلينا النظر فيه أثناء اتخاذ قراراتنا. نحن نختار الأكل الآن لأنَّه سيؤدي لمستقبلٍ أفضل، ولكن ليس بالضرورة أن نتنبأ بذلك المستقبل، فليس علينا مثلًا قول شي كهذا: "سآكل الآن لأنَّه عليّ إنقاذ طفلي غَدًا". لا يهم ما سيَحدُث غدًا، ما يُهم أنَّ الغد سيكون أفضل إن أكلت ولم تضوَّر جوعًا اليوم.

يمكننا في تصميم البرمجيّات، بمقياس مُشابه، اتخاذ قرارات معينة بناءً على معلومات نملكها الآن بهدف جعل المُستقبَل أفضل (تخفيض جهد الصيانة وزيادة القيمة)، وبدون الحاجة للتنبؤ بما سيحدُث مُستقبلًا. هناك استثناءات قليلة لهذا، فمثلًا أحيانًا تكون على دراية دقيقة بما سيَحدُث في المُستقبل القريب، فتستطيع عندها اتخاذ قرارات بناءً على درايتك هذه. ولكن إن كنت ستتخذ قرارًا بناءً على شيءٍ كهذا، سيتوجب عليك أن تكون مُتيقنًا من ذلك المُستقبَل، وينبغي أن يكون ذلك المُستقبل قريبًا. لا يهم مدى ذكاؤك، لأنَّه وببساطة يستحيل التنبؤ بدقّة بالمستقبل البعيد.

دعنا نأخذ مثالًا من خارج عالم البرمجة: الأقراص المُدمجة، التي صُمِمَت عام 1979م لاستبدال أشرطة الكاسيت القديمة لتكون طريقة رئيسيّة للاستماع للموسيقى. مَنْ كان يتوقَّع أنَّ بعد عشرين عام من ذلك ستُصنَع أقراص الفيديو الرقميّة بنفس شكل وحجم الأقراص المدمجة ليتمكَّن المصنعون من صناعة سوَّاقات للأقراص المُدمجة وأقراص الفيديو الرقميَّة للحواسيب؟ ومَنْ كان يتوقَّع مشاكل إدارة القرص

المُدمَج خمسين مرَّة أسرع مما كان مُفترَض له أن يدور عندما كان يُقرأ في "سواقات ذاكر القراءة فقط للقرص المُدمج"؟

لهذا في كل أنواع الهندسات، بما في ذلك مجال تطوير البرمجيّات، لدينا "مبادئ توجيهيّة". هذه المبادئ هي عبارة عن قواعد عند اتباعها تُبقي الأمور تعمل جيدًا بغض النظر عن ما هو آتٍ في المُستقبل. هذه هي قواعد وقوانين تصميم البرمجيّات، أي هي "مبادئنا التوجيهيّة" كمُصميمن.

نعم، من المهم التذكُّر أنَّ هناك مُستقبلًا، ولكن ذلك لا يعني أنَّ عليك تنبؤه. بل، عوضًا عن ذلك، عليك اتخاذ قراراتك استنادًا على القوانين والقواعد في هذا الكتاب؛ كونها ستقودُك إلى برمجيَّة بمستقبل أفضل بغض النظر عن ما هو آتٍ في ذلك المُستقبل.

يستحيل التنبؤ بكل الطرق التي قد يُساعدك بها قانون أو قاعدة مُعينة في المُستقبَل، ولكنَّه بالتأكيد سيفعل وستكون شاكرًا أنَّك طبقته على عملك.

مُرحَب بك ألاّ تتفق مع القوانين والقواعد والحقائق التي ستقرأها هنا، وأتمنى أن تستخلص استنتاجاتك الخاصة عنهم، ولكن تذكَّر أنَّك إن لم تتبعهم ستَقع غالبًا في الكثير من المشاكل في مُستقبَلٍ لا يُمكنك تكهنه.

الفصل الخامس: التغيير

والآن بعد أن فهمنا أهميّة المستقبل وفهمنا أنَّ به بعض الأمور التي لا نعرفها ولا يمكننا معرفتها عنه، ما الأمور التي يمكننا معرفتها عنه؟

إحدى الأمور الأكيدة هي أنَّ بيئة برمجيّتك ستتغيّر مع مرور الزمن. فلن يبقى شيء على حاله للأبد، وهذا يعني أنّ على برمجيّتك أن تتغيّر أيضًا لتتناسب مع البيئة من حولها.

وهذا ما يقودنا إلى "قانون التغيير":

كلما طالت مُدّة وجود برنامجك، كلما زادت احتماليّة وجوب تغيير أيّ جزءٍ منه مستقبلًا.

مع مرور الزمن إلى مستقبل لانهائي، فإنّ احتماليَّة وجوب تغيير كل جزء من برنامجك ستؤول إلى 100%. خلال الدقائق الخمس القادمة لن تضطر غالبًا إلى تغيير أي جزءٍ من برنامجك، وربّما قد تضطر إلى ذلك خلال الأيّام العشرة القادمة، أمّا خلال العشرين سنة القادمة فستضطر غالبًا إلى تغيير أغلبه (إنْ لم يكن كُلّه).

من الصّعب أن تتنبّأ ما الذي سَيُغيّر تمامًا ولماذا. ربّما تكون قد كتبتَ برنامجًا لقيادة سيّارة بأربع عجلات، ولكن أصبح الجميع يقود شاحنة بثمان عشرة عجلة مستقبلًا. أو ربّما تكون قد كتبتَ برنامجًا لطلّاب المدارس الثانويّة، ولكنّ التعليم الثانوي ساءَ جدًّا لدرجة أنَّ الطلاب لم يعودوا يفهمونه.

المقصود أنّه ليس عليك أن تحاول التنبؤ بما سيتغيّر. كل ما عليك فعله هو أن تضع في بالك أنَّ الأمور ستتغيّر، ولذلك اكتب برمجيّاتك بشكل مرن قَدَرَ الإمكان، وحينها ستتمكّن من ملاءمتها مع التغييرات المستقبليَّة مهما كانت.

التغيير في برنامج واقعي

دعنا نلقي نظرة على بيانات توضِّح تغيّر برنامج مع مرور الزمن. توجد مئات الملفات في هذا البرنامج، ولكن الصفحة هنا لن تتسع لتفاصيل كل هذه الملفّات، ولهذا أُختيرَت أربع ملفّات كأمثلة، وتفاصيلها موجودة في الجدول أدناه.

	الملف الأوّل	الملف الثاني	الملف الثالث	الملف الرابع
الفترة المُحلَّلة	خمس سنوات وشهرين	ثماني سنوات وثلاث أشهر	ثلاث عشرة سنة وثلاث أشهر	ثلاث عشرة سنة وأربع أشهر
الأسطر الأصليّة	423	192	227	309
الأسطر الثابتة	271	101	4	8
الأسطر الحاليّة	664	948	388	414
مُعدّل الزيادة	241	756	161	105
مرّات التغيير	47	99	194	459
الأسطر المُضافة	396	1026	913	3828
الأسطر المُزالة	155	270	752	3723
الأسطر المُعدَّلة	124	413	1382	3556
التغييرات الكليّة	675	1709	3047	11107
مُعدّل التغيير	1.6x	8.9x	13x	36x

يعرض الجدول أعلاه ما يلي:

- **الفترة المُحلَّلة**

الفترة الزمنية التي تواجد خلالها الملف.

- **الأسطر الأصليّة**

عدد الأسطر التي كانت مكتوبة عند إنشاء الملف.

- **الأسطر الثابتة**

عدد الأسطر التي مازالت كما هي دون تغيير منذُ إنشاء الملف.

- **الأسطر الحاليّة**

عدد الأسطر الموجودة الآن، عند نهاية الفترة المُحلَّلة، في الملف.

- **مقدار الزيادة**

الفارق بين "الأسطر الحاليّة" و"الأسطر الأصليّة".

- **مرّات التغيير**

العدد الإجمالي للمرات التي أجرى فيها المبرمج مجموعة من التغييرات على الملف (حيث تتضمن مجموعة التغييرات الواحدة تغييرات على عدة أسطر). تُمثِّل عادةً مجموعة التغييرات الواحدة إصلاحًا لعلّة وإضافة لميزة جديدة ...إلخ.

- **الأسطر المُضافة**

عدد المرّات التي أُضيفَ فيها سطرٌ جديد خلال فترة تواجد الملف بشكل عام.

- **الأسطر المُزالة**

عدد المرّات التي حُذِفَ فيها سطرٌ موجود خلال فترة تواجد الملف بشكل عام.

- **الأسطر المُعدَّلَة**

عدد المرّات التي تغيَّر فيها سطر موجود (دون حذفه أو إضافة واحد جديد بدلًا منه) خلال فترة تواجد الملف بشكل عام.

- **التغييرات الكليّة**

مجموع عدد "الأسطر المُضافة" و"الأسطر المُزالة" و"الأسطر المُعدَّلَة" في الملف.

- **معدّل التغيير**

مقدار زيادة قيمة "التغييرات الكليّة" على قيمة "الأسطر الأصليّة".

كلمة "أسطر" في الوصف أعلاه تشمل كل سطر في الملفات، بما فيها: الأكواد والتعليقات والتوثيقات والأسطر الفارغة. إذا حلّلتَ دون احتساب التعليقات والتوثيقات والأسطر الفارغة، فالفرق الأساسي الذي ستراه هو أنّ عدد "الأسطر الثابتة" سيَصغُر مقارنةً بباقي الأعداد. بعبارة أخرى، "الأسطر الثابتة" غالبًا ما تكون عبارة عن تعليقات وتوثيقات وأسطر فارغة.

أهم شيء ينبغي إدراكه من الجدول هو أنّ الكثير من التغييرات تُجرى على المشروع البرمجي. من المحتمل جدًّا أن يتغيّر أي سطر في الكود مع مرور الزمن، ولكنَّك لن تستطيع التنبؤ بما سيتغيَّر ومتى سيتغيَّر وكمية التغيير. كل ملف من الأربع ملفات تغيَّر بطريقة مختلفة عن الآخر (يمكنك ملاحظة ذلك بالنظر إلى الأعداد فقط)، ولكنهم تغيّروا جميعًا تغييرات كبيرة.

توجد كذلك بضعة أمور أخرى مثيرة بشأن الأعداد التي حصلنا عليها في التحليل:

- يُمكننا الرؤية بالنظر إلى مُعدَّل التغيير أنّ العمل على تغيير الملفات كان أكبر من العمل على كتابتها بدايةً. من المؤكَّد أنَّ عدد الأسطر ليس تقديرًا مثاليًّا للعمل الذي تمَّ حقًّا، ولكنه يعطينا على الأقل فكرة عامة عمّا حصل. أحيانًا يكون المُعدَّل ضخمًا، فمثلًا الملف الرابع غُيِّر 36 مرَّة ضعف عدد أسطره الأصليّة.
- عدد الأسطر الثابتة في كل ملف صغير مقارنةً بعدد "الأسطر الأصلية"، وأصغر حتى بالمقارنة مع عدد "الأسطر الحالية".
- الكثير من التغييرات حصلت على الملفات حتى وإن كانت تنمو ببطئ مع مرور الوقت. فمثلًا، نما الملف الثالث بمقدار 161 سطر فقط على مدار 13 سنة، ولكن خلال هذا الوقت بلغ العدد الكلي للتغييرات 3,047 سطر.
- عدد التغييرات الكليَّة دائمًا أكبر من عدد الأسطر الحالية. بعبارة أخرى، الأكثر احتمالًا أن تجد سطرًا مُغيّرًا في الملف من أن تجد سطرًا أصليًّا ما إن يكون الملف متواجدًا لفترة طويلة كافية.

- عدد الأسطر المُعدَّلة في الملف الثالث أكبر من عدد الأسطر الأصليّة زائد عدد الأسطر المضافة. عُدِّلَت أسطر الملف أكثر مما أُضيفَت له أسطر جديدة. بعبارة أخرى، غُيِّرَت الكثير من الأسطر مرارًا وتكرارًا. هذا شائِع الحدوث في المشاريع طويلة العمر.

لا تُشكِّل النقاط أعلاه كل الأمور التي يُمكِن تعلُّمها هنا، فهناك الكثير من التحليلات المثيرة الأخرى المُمكن استنتاجها من هذه الأعداد. نُشجعك على التعمُّق في هذه البيانات (أو العمل على أعداد مشابهة من مشروعك) ورؤية ما يمكنك تعلمه أيضًا.

يُمكنك إجراء تجربة تعلميّة جيدة أخرى عبر مُطالعة سجل التغييرات المُجراة على ملف مُعين. إذا كان لديك سجل لكل التغييرات المجراة على ملفات برنامجك، وكان لديك ملف واحد موجود منذ وقتٍ طويل، حاول مطالعة كل تغيير أُجري عليه منذ نشوئه. فكِّر فيما إذا كنت قادرًا على التنبؤ بهذا التغيير عندما كُتِب الملف، وفكِّر فيما إذا كان يُمكِن كتابة الملف بشكل أفضل لتسهيل ذلك التغيير. حاول بشكل عام فهم كُل تغيير أُجري ورؤية ما إذا كان بإمكانك تعلُّم أي شيء جديد عن تطوير البرمجيّات عبر قيامك بذلك.

الأخطاء الثلاث

هناك ثلاث أخطاء شائعة يقَع فيها المطورون عند محاولتهم التعامل مع "قانون التغيير"، وهي مرتبة أدناه حسب درجة شيوعها:

1. كتابة كود لا حاجة له.
2. عدم الاهتمام بجعل الكود سهل التغيير.
3. الكتابة بعموميّة مبالَغ فيها.

كتابة كود لا حاجة له

هناك قاعدة مشهورة في عالم تصميم البرمجيّات اليوم تُدعى "لن تحتاج إليه". تقول هذه القاعدة أنَّه لا ينبغي أن تَكتُب كودًا قبل أن تحتاج إليه حقًا. هذه القاعدة قاعدة جيدة لكن ربما اسمها غير لائق. فقد تحتاج إلى ذلك الكود في المستقبل، ولكن، وبما أنك لا تستطيع التنبؤ بالمستقبل، فأنت لا تعرف بعد الكيفيَّة التي ينبغي أن يعمل بها ذلك الكود لتُصمِّمه تبعًا لها. إذا كتبته الآن، أي قبل أن تحتاجه حقًا، فإنَّك حتمًا ستعيد تصميمه عندما تحتاج حقًا إليه. ولذلك اقتصد الوقت الذي كنت ستُضيّعُه في إعادة تصميم الكود مُجدَّدًا، وانتظر بكل بساطة وقت الحاجة إليه واكتبه في ذلك الحين.

هناك خطر آخر ناتج عن كتابة كود قبل الحاجة إليه، وهو أنَّ ذلك الكود غير المُستعمَل قد يؤدي إلى تعَفُّن البرمجيّة. حيث بما أنَّ ذلك الكود لا يُستعَمل إطلاقًا حاليًّا، فإنَّه قد يصبح مع مرور الوقت غير متزامن مع بقية نظامك، وهكذا فقد يؤدي لِعِلَلٍ لن تعلم بها. وعندما ستبدأ باستعماله، ستحتاج إلى قضاء بعض الوقت في محاولة إصلاحه، أو قد يَحدُث ما هو أسوأ، فقد تعتقد أنه خالٍ من العلل فلا تتحقق منه؛ مما قد يجعله سببًا في ظهور الكثير من المشاكل للمُستخدمين. يجب حقيقةً توسيع تلك القاعدة هنا لتُصبِح:

لا تكتب كودًا حتّى تحتاج فعليًّا إليه، واحذف أي كود غير مُستخدَم.

ومِن مقولتنا عليك كذلك التخلُّص من أي كود لم تَعُد بحاجةٍ إليه. ولا تنسى أنَّه يمكنك دائمًا إعادة إضافته إن احتجتَ إليه من جديد في وقتٍ لاحِق.

هناك الكثير من الأسباب التي تدفع الناس إلى الاعتقاد أنَّهم عليهم كتابة الكود قبل الحاجة إليه، أو حتى الاحتفاظ بكود غير مُستخدَم. أولها اعتقادهم أنَّهم قادرون على الالتفاف حول "قانون التغيير"، وذلك ببرمجة كل خاصيّة قد يحتاج إليها أي مُستخدِم الآن وعندها لن يكون البرنامج بحاجة إلى أيَّة تغيير أو تحسين في المستقبل. لكن ذلك خطأ، فمن غير الممكن كتابة نظام لا يتغيَّر طالما استمرَّ في امتلاك مستخدمين.

هناك آخرون يؤمنون أنَّهم يربحون بعض الوقت في المستقبل من خلال إنجاز بعض الأعمال الإضافيَّة الآن. تنجح أحيانًا هذه الفلسفة، ولكن ليس عندما تكتب أكوادًا غير ضرورية. حتى وإن كانت هذه الأكواد ستُصبِح ضرورية مُستقبلًا، فمن المؤكد أنَّك ستضطر إلى قضاء وقت في إعادة تصميمها، إذًا فأنت تُضيِّع وقتك.

كتابة أكواد غير ضروريّة: مثال من الواقع

في يومٍ من الأيام، أعتقدَ أحد المطورين - لندعه **(ماكس)** - مُخطِئًا أنَّه قادر على تجاهل هذه القاعدة. في برنامجه، كانت هناك قوائم منسدلة حيث يمكن للمستخدمين اختيار قيمة. كل شركة تستخدم هذا البرنامج بإمكانها تخصيص قائمة الخيارات المعروضة في كل قائمة منسدلة. بعض الشركات قد ترغب في أن تكون الخيارات عبارة عن أسماء ألوان، وأخرى قد ترغب في أن تكون أسماء مدن. قد تكون أي شيء. إذًا فقائمة الخيارات الصّالحة يجب أن تُخزَّن في مكانٍ ما بحيث تستطيع كل شركة تغييرها.

الأمر الواضح الذي كان يجب القيام به هو تخزين قائمة القيم، ولا شيء غير ذلك، فهذا كل ما كان مطلوبًا. لكن **(ماكس)** قرّر تخزين شيئين: قائمة القيم، وكذلك معلومات حول إذا ما كانت كل قيمة "نشطة" حاليًّا، أي إذا كان المستخدمون قادرين حاليًّا على اختيار هذه القيمة أم أنَّها معطّلة مؤقتًا.

على أية حال، (ماكس) لم يكتب أيّة أكواد لاستخدام المعلومات حول ما إذا كان كل حقل نشطًا أم لا. كلّ الخيارات كانت نشطة، كل الوقت، بغض النظر عمّا تقوله البيانات المُخزَّنة. كان على يقين من أنَّه سيكتب كودًا لاستخدام تلك البيانات (ولربما كان يظن حتى أنَّه سيفعل ذلك غدًا).

مرّت عدة سنوات، ولم تُكتب أكواد لاستعمال تلك البيانات. البيانات بقيت هناك، غير مستعملة، تُحيِّر المُستخدمين وتتسبَّب بالعلل. العديد من الزبائِن والمطورين كتبوا إلى (ماكس)، مُتسائلين لماذا لا يحدث شيء عندما يحاولون تغيير قائمة القيم يدويًا وتعيين الخيارات بأنَّها غير نشطة. افترض أحد المطورين بشكل خاطئ أنَّ الحقل "نشط" كان قيد الاستخدام وكتب كودًا لاستخدامه، رغم أنَّ بقيَّة النظام بقي لم يستخدمه. أثَّر ما فعله ذلك المطوِّر على المُستخدمين، الذين بدؤوا بالتبليغ عن علل غريبة يحتاج تعقبها إلى الكثير من العمل.

أتى أحد المطورين نهايةً وقال: "اليوم سأقوم بتنفيذ القدرة على تعطيل الخيارات!". اكتشف ذلك المطوِّر عند عمله على الميزة أنَّ الحقل "نشط" لم يكن مُصممًّا بشكل يناسب احتياجاته، لذا كان عليه أن يقوم ببعض العمل لإعادة التصميم من أجل تنفيذ ميزته.

النتيجة النهائيّة: العديد من العلل، والكثير من الارتباك، وعمل إضافي للمطور الذي أحتاج للكود. وهذا يعد نسبيًّا انتهاكًا بسيطًا للقاعدة! الانتهاكات الجسيمة يمكن أن تكون لها عواقب أسوأ بكثير، كمواعيد تسليم فائتة وكوارث كبرى وربما حتى تدمير مشروعك بالكامل.

عدم الاهتمام بجعل الكود سهل التغيير

أحد أكبر قاتلي المشاريع البرمجيَّة هو ما نطلق عليه اسم "التصميم الجامد". هذا يحدث عندما يُصمِّم المبرمج أكواد يصعب تغييرها. هناك طريقتان للحصول على تصميم جامد:

1. القيام بالعديد من الافتراضات حول المستقبل.
2. كتابة أكواد بلا تصميم كافٍ.

مثال: القيام بالعديد من الافتراضات حول المستقبل

وكالة حكومية - لنسمّها مستشفى (فِتِرَن) - تريد أن تصنع برنامجًا، ولنطلق عليه اسم "نظام الرعاية الصحية". قرَّرت الوكالة، قبل تنفيذ النظام، كتابة توثيق توضِّح فيه بدقة كيفيَّة تنفيذ النظام بأكمله، والذي استغرق الوكالة عامًا كاملًا لإنجازه. أُتّخِذت في هذا التوثيق كل القرارات التي تخصّ النظام.

قضى المطورون ثلاث سنوات في برمجة النظام وفقًا لما جاء في التوثيق. وخلال عملهم اكتشفوا أنَّ التصميم الذي جاء في التوثيق متناقض وغير كامل وصعب التنفيذ. لكن المستشفى أخذ عامًا كاملًا من أجل كتابة هذا التوثيق، والمطورون ليسوا قادرين على انتظار عام آخر لتنقيحه. لذلك أنجزوا النظام، متّبعين ما جاء في التوثيق قدر المستطاع.

انتهى العمل على النظام وسُلِمَ للمستخدمين لأوّل مرّة. على أيَّة حال، تغيَّرت الأحوال في المستشفى في آخر أربع سنوات بشكل كبير، وعندما بدأ المستخدمون فعليًّا باستخدام برنامج الرعاية الصحية، أدركوا أنّهم يريدون شيئًا مختلفًا كليًّا. لكنَّ النظام مكوَّن من مئات الآلاف من الأكواد، كُلَّها مُصمَّمة بجمود وفقًا للتوثيق، أي ببساطة يحتاج تغييرها الى أشهر أو سنوات من العمل. ولذلك بدأ المستشفى بكتابة توثيق جديد خاص بنظام جديد، وبدأت العملية من جديد.

غلطة المستشفى كانت محاولة التنبؤ بالمستقبل، حيث افترضوا أنَّ كل القرارات التي اتّخذوها في التوثيق كانت صالحة للمستخدمين الحقيقيين، وأنَّها ستبقى صالحة عندما يصبح النظام جاهزًا. عندما وصل المستقبل، لم يكن تمامًا كما تنبؤوا، ونظامهم كان عبارة عن فشل بملايين الدولارات.

الحل الأمثل كان تحديد ميزة واحدة فقط، أو مجموعة صغيرة من الميزات، ومطالبة المطورين

بتنفيذها فورًا. عندها يمكن أن يكون هناك تواصل واختبارات للاستخداميّة بالتزامن مع التطوير. عند إنجاز وإتمام أوَّل مجموعة من الميزات، يمكنهم حينها العمل على ميزات أخرى إضافية، الواحدة تلو الأخرى، إلى حين حصولهم على نظام مُصمَّم بشكل جيد يلبي كل احتياجات مستخدميه.

مثال: كود بلا تصميم كافٍ

طُلِبَ من مُبرمِج إنشاء برنامج يُمكِّن المُستخدمين من متابعة المهام التي يفعلونها. لإنشاء "مهمة" في هذا النظام، يملأ المُستخدم استمارة ببعض المعلومات، كملخص قصير عن المهمة وما أنهى منها لحد الآن، تُخزَّن في قاعدة بيانات. يستطيع المُستخدِم بعدها ترك ملاحظات عن تَقدُمهِ في إنجاز هذه المهمة مع مرور الوقت، ونهايةً يُمكنه وضع ملاحظة تُشير أنَّه أكملها.

يوجد حقل في هذا البرنامج يدعى "الحالة". يُشير هذا الحقل إلى مدى تَقدُم المُستخدم في أداء المهمة. يأخذ هذا الحقل القيم: "لم يُنجَز شيء بعد" و"قيد التقدم" و"قيد الانتظار" و"اكتملت". عندما تكون قيمة الحقل "لم يُنجَز شيء بعد" لا يُمكن تغييرها سوى إلى "قيد التقدم"، وعندما تكون قيمة الحقل "قيد التقدم" لا يُمكن تغييرها سوى إلى "اكتملت"، وعندما تكون قيمة الحقل "اكتملت" لا يُمكن إرجاعها سوى إلى "قيد التقدم".

توجد عشرة حقول أُخرى في البرنامج لها قواعد مُشابهة، وكلٌ منها يحتوي معلومات مختلفة عن المهمة (كالِمَن أُسنِدَت والموعد النهائي لتسليمها).

كَتبَ المُبرمِج لتنفيذ هذه القواعد كود شديد الضخامة عديم الهيكلة وفي ملفٍ واحد. ربط مُبرمِج هذه البرمجيَّة كل حقل برمز مُخصَّص لإنجاز التوثُّق (validation)، فمثلًا في كل مرَّة كان يحتاج فيها للتحقُّق ما إذا كانت المُهمة "اكتملت"، كان يكتب الكلمة "اكتملت" (أو مرادفها

الإنجليزي "Complete" كونه يستخدم لغة برمجة إنجليزيّة) حرفيًّا في الكود، مما جعل الكود غير صالحًا لإعادة الاستخدام. نسخ المُبرمِج عندما رغِبَ بإنشاء حقول مشابهة نفس الكود وعدَّل عليه قليلًا لينجِز العمل المطلوب. اشتغلَ كود هذا المُبرمِج، فاقِد التصميم كليًّا تقريبًا، والمُحتوَى في ملف بطول 3,000 سطر.

ترك هذا المطوِّر المشروع بعد عدّة شهور لاحقة.

جاء مطوِّر آخر مكان السابِق وأُسنِدَت إليه وظيفة صيانة المشروع. اكتشف هذا المطوِّر بسرعة أنَّ الكود صعب التغيير، فإذا ما غيّر جزءًا واحدًا سيضطر لتغيير العديد من الإجزاء الأخرى ليُحافِظ على الكود مُشتغِلًا. وما زادَ الأمر سوءًا أنَّ أجزاء البرنامج مُبعثرة دون أي تفسير أو تنظيم منطقي، بشكل سيضطرُّك إلى قراءة الملف كاملًا كلما أردت إجراء تغيير.

بدأ الزبائِن يطلبون ميزات جديدة، وبدأ المُبرمِج يحاول تنفيذها. بذل المُبرمِج الجديد قصارى جهده محاولًا تنفيذ هذه الميّزات، وأضاف المزيد من الأكواد إلى هذا الملف، مما زاد ضخامته إلى أن وصل إلى 5,000 سطر.

بدأ الزبائن يطلبون في نهاية المطاف المزيد من الميّزات الغير مُمكِن تنفيذها بهذا التصميم الحالي. حيث أرادوا إرسال المعلومات عن المهام بالبريد الإلكتروني، وهو ما لا يتحمله الكود؛ كونَهُ مُصمَّم فقط وتحديدًا للعمل مع الاستمارات.

تفاقم الوضع وظهرت الكثير من البرمجيّات المنافسة ذات ميزات متفوقة، كتحديث المهام عبر البريد، مما جعل المشروع يخسر زبائنه تدريجيًّا.

السبب الوحيد وراء صمود المشروع هو قضاء مطوِريين سنة كاملة في إعادة تصميم هذا الملف وحسب ليصبِح تغييره سهلًا. فعل هاذان المطوران ما في وسعهما لتنفيذ الميّزات المطلوبة من

الزبائِن أثناء إعادة تصميمهم للمشروع، ولكن معظم الوقت هُدِر على إعادة التصميم[5].

القاعدة المُستخدمة لتجنب بناء تصاميم جامدة:

ينبغي أن يُصمَّم الكود بناءً على ما تعرفه الآن، وليس على ما تظن أنَّه سيحدُث مُستقبلًا.

تصميمك ينبغي أن يعتمد فقط على متطلباتك الحالية المعروفة دون استبعاد احتمال إضافة متطلبات مستقبليّة. إذا كنت مثلًا تحتاج إلى نظام برمجي يقوم بالوظيفة "س" وحسب، فعليك تصميمه لكي يقوم بها فقط حاليًا. قد يقوم هذا النظام بأشياء أخرى مُستقبلًا غير "س"، وهو ما عليك وضعه في ذهنك أثناء العمل عليه، ولكن حاليًّا على النظام أن يقوم بالوظيفة "س" فقط.

يُبقي التصميم بهذه الطريقة كذلك تغييراتك الفرديّة صَغيرة، مما يُسهِّل بناء تصاميمك عليها. ليس معنى قولنا أنَّ التخطيط سيء، فالتخطيط بقدر مُعين شديد القيمة في تصميم البرمجيَّات. ولكن حتى ولو لم تكتُب خطَّة مُفصلة ستبقى بخير طالما كانت تغييراتك دائمًا صغيرة وكودك السهل التكيُّف مع المُستقبل المجهول.

الكتابة بعموميّة مبالَغ فيها

يحاول بعض المبرمجون عندما مواجهتهم حقيقة أنَّ كودهم سيتغيّر مُستقبلًا حل المُشكلة عبر تصميم حل مبالغ فيه **يعتقدون** أنَّه سيستوعب كل وضع مُستقبلي مُمكن. ندعو هكذا فعل في عالم تصميم البرمجيّات "بالهندسة الفائقة".

5. هذه قصة الملف *process_bug.cgi* من مشروع (بجزيلا). القصة أعلاه نسخة مُبسطة لِما حدثَ في الحقيقة، إلا أنَّ الأرقام المذكورة (كعدد الأسطر البرمجيّة والوقت الذي قُضيَ في إعادة التصميم) دقيقة تمامًا. طالِع سِجلات المشروع لرؤية تاريخ إعادة تصميمه كاملًا.

يُعرِّف القاموس "الهندسة" بالتصميم والتخطيط، و"الفائقة" الفاعل المؤنَّث من "فاق" بالزيادة في التفوق والامتياز. معنى "الهندسة الفائقة" اصطلاحًا تصميم أو بناء شيء أكثر مما تحتاج في موقفك.

ماذا نعني بالبناء والتصميم **أكثر مما تحتاج**؟ ماذا تعني عبارة "أكثر مما تحتاج" هنا؟ أليس التصميم شيئًا جيدًا؟

صحيح أنَّ معظم المشاريع ستستفيد من تخطيط وتصميم أكثر، كما رأينا في مثال "كود بلا تصميم كافٍ"، إلا أنّه أحيانًا قد يُبالِغ الشخص بالأمر، كأن يبني سلاح ليزر مداري لتدمير عش نمل صغير. جهاز الليزر المداري تصميم هندسي مُذهِل، ولكنَّهُ يُكلِّف قدرًا هائِلًا من المال ويستَغرِق وقتًا طويلًا لإنجازه وذلك بعيدًا عن أنَّ صيانته كابوس. هل لك أن تتخيَّل الصعود إلى الفضاء لإصلاحه إذا ما انكسر؟

توجد عدّة مشاكل أخرى بشأن الهندسة الفائقة:

1. لا يُمكنك التبنؤ بالمُستقبَل مهما كان حلُّك عامًّا، فهو لن يكون عامًّا كفاية لتلبية متطلباتك المستقبلية الحقيقية.
2. كتابة كودك بعموميّة مبالغة سيجعله غالبًا سيئًا في التعامل مع الأشياء المُحدَّدة من وجهة نظر المُستخدِم. لنقل مثلًا أنَّك تُصمِّم كودًا يعامل كل المُدخلات بنفس الطريقة على أنَّها بايتات. سيُعالِج أحيانًا هذا الكود نصوصًا وأحيانًا أخرى صورًا، ولكن كل ما يعرفه أنَّه يستقبل بايتات. تلك طريقة جيدة في التصميم، فالكود المُنتَج بسيط وسهل ومُستقل.

   ولكن عدم جعل كودك قادرًا على التفريق بين النصوص والصور شيء عام بشكل مبالَغ. فمثلًا إذا ما مرَّرَ المُستخدِم صورةً فاسدة سيُخبره الكود: "لقدر مرَّرَت بايتات فاسدة" بدلًا من "لقد مرَّرتَ صورةً فاسدة"؛ لأنَّه مكتوب بشكل عام جدًّا ليستطيع إخباره الخطأ بدقة (هناك العديد من الطرق الأخرى التي يفشَل فيها الكود من هذا النوع عند التحدُّث عن استخدامات مُحدَّدة وهذا مُجرَّد مثال).

3. يتطلَّب العمل بطريقة عامة كتابة الكثير من الأكواد الغير ضروريّة، مما يُعيدنا للخطأ الأوَّل (كتابة كود لا حاجة له).

إذا ما كُنتَ عامةً تَصنَع أشياء مُعقدة عوضًا عن محاولة تبسيطها فأنت تُهندِس بفواقة. سيُعقِّد جهاز الليزر المداري ذاك حياة مَنْ يرغب بتدمير بعض أعشاش النمل وحسب، بينما سُم بسيط سيُسهِّل حياته ويُزيل مشكلة النمل.

كُونك عامًّا بالوقت المناسب وبالطريقة المناسبة قد يكون أساسًا لتصميم برمجيّات ناجحة، ولكن كونك شديد العموميّة قد يُسبِّب تعقيد وتشتيت وجهود صيانة مهدورة. تتشابه قاعدة تفادي العموميَّة الزائدة مع قاعدة تفادي التصاميم الجامدة، حيث تقول:

كُن عامًّا بقدر ما تحتاج أن تكون الآن.

مثال: عموميّة مبالغ فيها!

في أحد أجزاء برنامج، كان المُستخدِمون يملؤون استمارة ليُرسِل البرنامج بضعة مِئاتٍ من البُرُد الإلكترونيّة. كان ذلك الجزء بطيئًا جدًّا، لدرجة أنَّ المُستخدِم كان يُرسِل الاستمارة وينتظر البرنامج وقتًا طويلًا ليُنهي عمله.

قرَّرَ مُطورو البرنامج، سعيًا لجلعه أسرع، عدم إرسال كل البُرُد فورًا، بل إرسالها في الخلفية بعدما يُرسِل المُستخدِم الاستمارة باستخدام جزئيّة كود مكتوبة سلفًا تُدعى "مُرسِل البريد".

بدأ المطوِّر المسؤول عن هذا التغيير العمل عليه مُقرِّرًا أنَّ بعض الشركات قد ترغَب باستخدام مُرسِل بريد آخر. كتب هذا المطوَّر المئِات من الأسطر ليُمكِّن الزبائِن من "إلحاق" أنظمة أخرى للقيام بالعمل في الخلفيّة. لم يطلُب أحد من الزبائِن ذلك، ولكنَّ المطوِّر تبنًّا أنَّ أحدهم قد يطلُب هذه المرونة في المُستقبَل.

استلم رئيس مُهندسي البرنامج نهايةً هذا التغيير، وأزال كل الأكواد المسؤولة عن إمكانيَّة "إلحاق" أنظمة أخرى؛ وذلك لعدم وجود ما يدُل على أنَّ المُستخدمين يرغبون بهذه الخاصيّة، وبالتالي لا دليل على وجوب كون الكود بهذه العموميَّة حاليًّا. مع إزالة هذه الأجزاء غدا التغيير أبسط بكثير من ذي قبل.

مرَّت أربع سنوات على إجراء هذا التغيير، ولم يَحتَج زبونٌ واحد لهذه الخاصيّة. مما يوضِّح أنَّه لم يكن من داعٍ لكل هذه العموميّة.

التصميم والتطوير التصاعدي

هناك طريقة في تطوير البرمجيّات تُعين على تجنُّب الأخطاء الثلاث السالف ذكرها تُدعى "بالتصميم والتطوير التصاعدي". تعمل هذه الطريقة على مبدأ بناء وتصميم النظام قطعة قطعة على حدة بالترتيب بشكل تصاعدي (من الخطوة الأسهل صعودًا إلى الخطوة الأصعب).

من الأسهل شرح آلية عمل هذه الطريقة بمثال، وهذا ما سنفعله. يُظهِر المثال أدناه كيفيَّة استخدام هذه الطريقة لتطوير آلة حاسبة تُجري عمليات الجمع والطرح والضرب والقسمة:

1. خطِّط لنظام يُجري عملية الجمع وحسب.
2. نفِّذ ذلك النظام.
3. أصلح تصميمه ليُصبِح من السهل إضافة ميزة الطرح إليه.
4. نفِّذ ميزة الطرح في النظام. أصبح الآن لدينا نِظام يُجري عملية الجمع والطرح فقط.
5. أصلح تصميمه ليُصبِح من السهل إضافة ميزة الضرب إليه.
6. نفِّذ ميزة الضرب في النظام. أصبح الآن لدينا نظام يُجري عملية الجمع والطرح والضرب فقط.
7. أصلح تصميمه مُجدّدًا ليصبح من السهل إضافة ميزة القسمة إليه (سيستغرِق الأمر في هذه النقطة جهدًا أقل أو ربما لن يستغرق شيئًا؛ كوننا حسَّنّا التصميم قبل تنفيذ الطرح والضرب).

8. نفِّذ ميزة القسمة في النظام. الآن حصلنا على النظام الذي بدأنا ببنائه، مع تصميم ممتاز مناسب له.

تتطلَّب هذه الطريقة في التطوير وقتًا وتفكيرًا أقل من التخطيط المُسبَق للنظام كاملًا وبناءه دفعة واحدة. قد لا يكون الأمر سهلًا إذا ما كُنتَ قد اعتدتَ طرق تطويريَّة أخرى في البداية، ولكنه سيُصبِح أسهل بالممارسة.

الجزء الصعب في هذه الطريقة هو تقرير ترتيب تنفيذ المهام. عليك عمومًا اختيار أسهل ميزة والعمل عليها في كل خطوة. اخترنا ميزة الجمع أوَّلًا كونها الأسهل من بين بقيَّة العمليات، والطرح ثانيًا لإمكانيَّة بنائها منطقيًّا على الجمع بسهولة تامة. كان بإمكاننا أيضًا اختيار عملية الضرب ثانيًا كذلك؛ حيث أنَّها عبارة عن الجَمِع عدِّة مرَّات. العملية الوحيدة التي ما كان بإمكاننا اختيارها ثانيًا هي القسمة؛ كون الانتقال من الجمع إلى القسمة مباشرةً قفزة منطقيَّة كبيرة نظرًا لتعقيد القسمة بالمقارنة. بينما الانتقال من الضرب إلى القسمة في آخر خطوة كان سهلًا للغاية، مما جعله اختيارًا موفقًا.

أحيانًا لربما تحتاج إلى أخذ ميزة واحدة وتقسيمها إلى خطوات منطقيَّة أصغر وأبسط لتتمكَّن من تنفيذها بسهولة.

هذه الطريقة فعليًّا عبارة عن توليفة من طريقتين: الأولى تُدعى "التطوير التصاعدي" والثانية "التصميم التصاعدي". التطوير التصاعُدي هو طريقة لبناء نظام مُتكامِل عبر تقسيم العمل إلى أجزاء أصغر، بينما التصميم التصاعدي هو طريقة لإنشاء وتحسين تصميم النظام بخطوات تصاعديّة صغيرة. في قائِمتنا، الخطوات التي تبدأ بـ"نفِّذ" تُمثِّل جزءًا من عملية التطوير التصاعدي، وتلك التي تبدأ بـ"أصلح تصميمه (الهاء تعود هنا للنظام)" أو "خطِّط" تُمثِّل جزءًا من عملية التصميم التصاعدي.

التطوير والتصميم التصاعدي ليس الطريقة الوحيدة لتطوير البرمجيات، لكنها طريقة فعالة تضمن تجنُّب القوانين الثلاثة المذكورة سابقًا.

الفصل السادس: العيوب والتصميم

لا يوجد مُبرمِج مثالي لسوء الحظ. المبرمجون الجيدون سيحتوي كودهم على عيب في كل 100 سطر من الكود تقريبًا. أما أفضل المبرمجين في أفضل الظروف سيحتوي كودهم على عيب في كل 1000 سطر من الكود.

بعبارة أخرى، بغض النظر عن مقدار خبرتك كمبرمج، من المؤكَّد أنك كلّما كتبت أكواد أكثر كلّما أنتجت عيوبًا أكثر، وهذا ما يقودنا إلى قانون يُسمّى "قانون احتماليّة العيب":

تتناسب فرصة أن تنتج عيبًا في برنامجك طردًا مع حجم التغييرات التي تجريها عليه.

وهو مهم، لأنَّ العيوب تُضِرّ بهدفنا من مساعدة الناس، ولهذا يجب أن تُتجنَّب هذه العيوب. وكما أنّ إصلاح العيوب يُعتبر شكل من أشكال الصيانة. وبالتالي، كلما زاد عدد العيوب زاد الجهد المبذول في الصيانة.

مع هذا القانون، ودون الحاجة إلى التنبؤ بالمستقبل، بإمكاننا أن نرى مباشرةً أنَّ إجراء تغييرات صغيرة سيؤدي إلى جهد أقل في الصيانة مقارنةً بإجراء تغييرات كبيرة. تغييرات صغيرة = عيوب أقل = صيانة أقل.

يُعبَّر أحيانًا عن هذا القانون بالتالي: " لن تستطيع أن تُحدِث عِلَلًا برمجيَّة جديدة ما لم تُعدّل أو تضيف أكوادًا".

الفكاهي بشأن هذا القانون أنَّه يبدو وكأنَّه يتضارب مع قانون التغيير الذي يقول أنَّ على برمجيّتك أن تتغيّر، ولكن تغييرها حسب هذا القانون سيؤدي لإنتاج عيوب. هذا تضارب حقيقي، وموازنة هذا التضارب بين هذين القانونيين تتطلب ذكاءك كمصمم برمجيَّات. هذا التضارب هو ما يُبيّن حقيقةً سبب حاجتنا إلى التصميم، ويخبرنا بماهيّة التصميم المثالي:

أفضل تصميم هو ذاك الذي يسمح بأكبر تغيير في البيئة مع أقل تغيير في البرمجيَّة.

وببساطة هذا يلخّص الكثير مما نعرفه اليوم عن تصميم البرمجيات الجيدة.

لا تصلحه ما لم يكن مكسورًا

عرفنا الآن أنَّه لا يُمكننا إدخال علل إلى برنامجنا ما لم نُضف على أو نُعدِّل كوده، والذي هو قانون هام في علم تصميم البرمجيَّات. توجد كذلك قاعدة أخرى شديدة الأهميَّة مُرتبطة بالسابقة يسمعها الكثير من مُهندسي البرمجيّات بصيغة أو أخرى، تقول:

لا "تُصلِح" أي شيء ما لم يكن مُشكلةً، ولا تفعل حتى تملك دليل يؤكِّد أنَّه كذلك.

من المُهم جدًا أن تملك دليلًا أنَّ المشاكل موجودة قبل أن تُعالجها، وإلّا قد تطوِّر ميزةً لا تحل مشكلة لأي أحد، أو قد "تُصلِح" أشياءً ليست مكسورة.

إذا ما كنت تُصلِح مشاكل بدون دليل على وجودها فأنت غالبًا ستكسر أشياءً في برنامجك بدلًا من إصلاحه. أنت تُدخِل تغييرات جديدة على نظامك، والتي ستُدخِل بدورها عيوبًا جديدة. وليس هذا فحسب، بل أنت تَهدُر وقتك أيضًا على إضافة تعقيد لبرنامجك بلا أي سبب.

فما الذي يُحسَب "كدليل"؟ لنفترض أنَّ خمسة مُستخدمين بلَّغوا أنَّهم عندما يضغطون على الزر الأحمر البرنامج ينهار. ذلك يُعتبر دليل كافٍ، أو حتّى لو ضغطتَ بنفسك على الزر الأحمر ولاحظتَ أنّ البرنامج ينهار فهذا دليل.

لمُجرَّد أنَّ المُستخدِم بلَّغ عن شيء لا يعني أنَّه مُشكلة. فلا يُدرِك المُستخدِم أحيانًا أنَّ البرنامج يملك ميزة ما مسبقًا، فيطلب منك أن تُنفِّذ شيء غير ضروري. لنأخذ مثالًا: لنقل أنَّك تكتب برنامجًا يُرتِّب قائمة من الكلمات هجائيًّا، ومُستخدِم لبرنامجك طلبَ منك أن تُضيف ميزة تُرتِّب قائمة من الحروف. برنامجك يُمكنه إنجاز هذا بالفعل، بل أكثر من هذا أحيانًا (هذا الذي يحصل غالبًا في حالات الطلبات المُحيِّرة كهذه). قد يظن مُستخدِم برنامجك أنَّ هناك مُشكلة في حالتنا بالرغم من عدم وجودها، بل وقد يُقدِّم

حتّى "دليلًا" على أنَّه لا يستطيع ترتيب قائمة حروف، في حين أنَّ المشكلة حقيقةً هي عدم إدراكه أنَّ عليه استخدام ميزة ترتيب الكلمات نفسها.

> عليك إصلاح الأمر إذا ما وردتك طلبات كثيرة كتلك؛ فذلك يعني أنَّ المُستخدمين غير قادرين على العثور على الميزات التي يحتاجونها في برنامجك.

قد يُبلِّغ المُستخدِم أحيانًا عن وجود علَّة في حين أنَّ البرنامج يعمل فعليًّا كما هو متوقَّع. في حالات كهذه القاعدة قاعدة أغلبيّة: فإذا كان عدد كبير من المُستخدمين يظنون أنَّها علّة فهي كذلك، وإذا كانت أقليَّة صغيرة (كشخص أو شخصين) تظن أنَّها علّة فهي ليست كذلك.

أكثر الأخطاء شيوعًا في هذه الناحية تُدعى "التحسين المُبكِّر". يعمل المطورون، في هذا الخطأ، على تسريع برنامجهم، ويهدرون وقتًا في تحسين كودهم قبل أن يعلموا حتى أنَّه بطيء! هم كالجمعيَّة الخيريَّة التي تُرسِل طعامًا للأغنياء قائلةً: "نعمل على مساعدة الناس"، أليس ذلك عديم المنطقيَّة؟! هم يحلُّون مشكلةً غير موجودة.

الأجزاء الوحيدة من البرنامج الواجب النظر في جعلها سريعة هي تلك التي تستطيع أن ترى بوضوح أنَّها تُسبِّب مشاكل في الأداء لمُستخدمي برنامجك. يجب أن تُركِّز على المرونة والسهولة في بقيَّة الأجزاء، لا على جعلها سريعة.

هناك طرق لا محدودة لكسر هذه القاعدة، ولكن هناك طريقة واحدة بسيطة لتتبعها: اجلُب دليلًا يؤكِّد أنَّ المُشكلة هي حقًّا مُشكلة قبل أن تُعالجها.

لا تُكرِّر نفسك

هذه غالبًا القاعدة الأكثر شُهرة في تصميم البرمجيّات. ليس هذا الكتاب أوَّل كتاب يطرحها، ولكنها قاعدة صحيحة ولهذا أُضيفَت هنا:

لا ينبغي أن تُضاف أي قطعة من المعلومات، في أي نظامٍ كان، مرَّتين.

لنقل أنَّك تملك حقلًا يُدعى "كلمة المرور" يظهر في مِئة نافذة من واجهة برنامجك الرسوميّة. ماذا لو أردت تغيير اسم الحقل مثلًا إلى "رمز المرور"؟ هذا يعتمد على طريقة كتابتك للكود: فإن كُنتَ قد خزَّنتَ اسم الحقل في مواقع واحد مركزي في كودك، فسيتطلَّب الأمر تغيير واحد وحسب. بينما إن كنت قد كتبتَ اسم الحقل يدويًّا في المئة نافذة، فسيتطلَّب الأمر تغيير مئة سطر لإصلاح المشكلة.

تنطبق القاعدة أيضًا على كتل الأكواد. لا ينبغي عليك نسخ الكتل، بل استخدام تقنيات البرمجة المُختلفة التي تُمكِّن قطعة من الكود "استخدام" أو "استدعاء" أو "ضَمْ" قطعة أخرى.

إحدى الأسباب الجيدة لاتباع هذه القاعدة هي قانون احتماليَّة العيب. إذا كان بإمكاننا إعادة استخدام كود قديم، فذلك يعني أننا لن نكتُب أو نُغيِّر الكثير من الأكواد عند إضافة ميِّزات جديدة، وبالتالي احتماليّة إدخال عيوب أقل.

تُساعد هذه القاعدة أيضًا على إضفاء المرونة على تصاميمنا. فإذا ما احتجنا، مثلًا، لتغيير كيفيَّة عمل برنامجنا، سيكون بإمكاننا تغيير بعض الأكواد في مكان واحد فقط عوضًا عن المرور على البرنامج كاملًا وإجراء عدِّة تغييرات.

الكثير من التصاميم الجيدة بُنيت على تلك القاعدة، والتي تعني: كلما كُنت أذكى في جعل أكوادك "تستخدم" أخرى وفي مَركزِة المعلومات، كُلَّما أنجزتَ تصاميمًا أفضل. هذه ناحية أخرى حيث ذكاءك يلعب دورًا في البرمجة.

الفصل السابع: البساطة

سوف نتفادى العيوب تمامًا إن لم نُغيّر برمجياتنا. لكن التغيير لا مفر منه، وخصوصًا عند إضافة ميزة جديدة. لذلك عدم تغيير شيء ليس بالتقنية الأمثل لتقليل العيوب.

كما شرحنا سابقًا في الفصل السادس، لتفادي العيوب يُستحسن أن تكون تغييراتنا طفيفة. ولكن إن أردت أن تتخلص من العيوب حتى من تغييراتك الطفيفة، هناك قانون آخر يمكن أن يساعدك. ولن يُقلل من العيوب فحسب، بل أيضًا سيحافظ على كودك مُصانًا، وسيُسهِّل من إضافة ميزات إليه، وسيُحسِّن مفهوميّته الكليّة. ذلك القانون يُسمَّى "بقانون البساطة":

تتناسب سهولة صيانة أي جزئيَّة من برمجيّتك مع بساطة أجزائِها الفرديّة.

وبالتالي، كلما بَسُطَت الأجزاء الداخليَّة، كُلما سَهُل تغيير الأمور في المستقبل. سهولة الصيانة المثالية مستحيلة، لكن ذلك ما تصبو إليه (تغيير شامل أو إضافات لانهائية من الأكواد دون صعوبة).

لربما انتبهت أنَّ هذا القانون لا يتطرق إلى بساطة النظام البرمجي بأكمله، بل أجزاءه الفرديّة. فلِمَ ذلك يا ترى؟ أي برنامج حاسوبي متوسط الحجم مُعقَّد لدرجة لا يستطيع بشري استيعابه دفعة واحدة. بالمقدور استيعاب أجزاء منه لا غير.

فدائمًا ما يكون برنامجنا ذو بنية واسعة ومعقدة وما يهم حينها هو فهم أجزاءه عندما ننظر إليها. كلما سَهُلَت الأجزاء كلّما سهل فهمها من الأشخاص. هذا يساعد خصوصًا عندما تُسلِّم كودك لأشخاص آخرين أو لمَّا تتوقف عن التكويد لبضعة أشهر وتعود فيما بعد لتستوعب ما قمت به.

مثال من الهندسة المعماريّة

تخيّل أنَّك تبني هيكلًا فولاذيًّا طوله 30 قدمًا. تستطيع أن تستعمل مجموعة من العوارض الصغيرة، أو أن تصنع ثلاث قطع فولاذيَّة ضخمة ومعقدة وتبني بهم هيكلك. يَسهُل شراء أو صنع العوارض، ولما تنكسر عارضة تستطيع استبدالها بأخرى، فبهذه الطريقة الإنشاء سهل وكذلك الصيانة.

من جهة أخرى، الثلاث قطع الضخمة يجب أن تُصنَع بعناية شديدة، وكل قطعة كاملة يَصعُب أن تجدها وتُصلح عيوبها بسبب كِبرها. وحتى عند الانتهاء من إنشاء المبنى، عندما ستكتشف العديد من الأخطاء فيه، لن تستطيع استبدال أي قطعة لأنَّ المبنى سيسقط إن فعلت. لذلك سيكون عليك لحمها بِرُقَع بشعة من الحديد وتتمنى أن لا يتهاوى المبنى.

الأمر مشابه في البرمجيات، فلما تكتب أجزاء بسيطة ومتكاملة من الكود، يسهُل إصلاح عيوب النظام وصيانته. لكن عند تصميم أجزاء واسعة ومعقدة، كل واحدة تتطلَّب جهدًا ولا تكون بالجودة المناسبة. بالتالي تَصعُب صيانة النظام، ولن تكُفَّ عن الترقيع كي يستمر في العمل.

لِمَاذا إذًا يكتب البعض البرمجيات كأجزاء واسعة وضخمة بدلًا من صغيرة وبسيطة؟ استعمال طريقة الأجزاء يوفِّر وقتًا مُعتبرًا عند بداية إنشاء البرمجية. لكن بطريقة الأجزاء الصغير يُستهلك الكثير من الوقت عند جمعها معًا، عكس الطريقة الأخرى حيث هناك القليل من الأجزاء سهلة الجمع معًا. على الرغم من ذلك، النظام المبني على القطع الكبيرة ذو جودة رديئة، وستقضي الكثير من الوقت في إصلاحه مستقبلًا، وتصعب صيانته مع الوقت، بينما تسهل في النظام البسيط. فعلى المدى البعيد البساطة فعالة، عكس التعقيد.

كيف سنستعمل إذًا هذا القانون في العالم التطبيقي للبرمجة؟ هذا الموضوع سيأخذ حقه في بقية الكتاب، مع ذلك عمومًا الفكرة هي جعل كل مُكوِّن من مُكوِّنات كودك بسيطًا قدر الإمكان، والتأكُّد من بقائه كذلك مع مرور الوقت.

إحدى الطرق الجيدة لفعل ذلك هي طريقة التصميم والتطوير التصاعدي المشروحة في نهاية الفصل الخامس. توجد في هذه الطريقة خطوة "إعادة التصميم" التي تأتي قبل أي إضافة الميزة، ويمكنك خلالها تبسيط النظام. وحتى إن لم تستعمل تلك الطريقة، يمكنك أن تُبسِّط الأجزاء التي تبدو صعبة لك أو لزملائك المبرمجين خلال الوقت الذي يفصل إضافة الميزات.

بطريقة أو بأخرى يجب دائمًا تبسيط ما أنشأته. لا يمكنك الاعتماد على أنَّ تصميمك الأولّي سيكون الأفضل دائمًا. عليك الاستمرار بإعادة تصميم نظامك عند ظهور أيّة حالات أو متطلبات جديدة.

قد تكون بالتأكيد هذه المهمة في غاية الصعوبة. لن تحصل دائمًا على أدوات بسيطة لكتابة برنامجك (فقد تحصل على لغات معقدة أو حاسوب بحد ذاته معقد)، لكن عليك السعي إلى التبسيط بما لديك.

البساطة ومعادلة تصميم البرمجيات

ربما لاحظت أنَّ هذا القانون يخبرنا أنّ أهم شيء يمكننا فعله حاليًا لتقليص جهد الصيانة في معادلة تصميم البرمجيات هو جعل الكود أبسط. لتحقيق ذلك ليس علينا أن نتنبأ بالمستقبل، يمكننا فقط النظر إلى الكود إن كان معقدًا لنجعله أبسط. العمل باستمرار لجعل كودك أبسط هذا ما يُمكِّنك من تقليص جهد الصيانة مع مرور الوقت.

هذا التبسيط يحتاج بعض العمل، لكن عمومًا فإنّ إحداث تغييرات على نظام بسيط أسهل بكثير منه في نظام معقد؛ لذلك خذ بعض الوقت لتبسيطه الآن من أجل ربح كثير من الوقت في المستقبل. عندما تُقلِّص جهد الصيانة الخاص بنظامك، فإنَّك تضاعف الرغبة في إحداث كل التغييرات الممكنة عليه (إن

كنت تريد إنعاش ذاكرتك لتذكر التفاصيل ارجع الى الفصل الرابع وألقِ نظرة أخرى على معادلة تصميم البرمجيات). تبسيط كودك يُقلِّل جهد الصيانة، ويضاعف بذلك الرغبة في إحداث أي تغيير ممكن.

البساطة نسبيّة

نحن نريد جعل الأمور أبسط، ولكن كيفيَّة تعريفك لكلمة "بسيط" مُتعلِّقة أساسًا بجُمهورك المُستهدَف. ما هو بسيط بالنسبة لك قد لا يكون بسيطًا بالنسبة لزملائك في العمل. وأيضًا، عندما تصنع شيئًا ما، فقد يبدو لك "بسيطًا" نسبيًّا لأنَّك تفهمه من الداخل والخارج، لكنَّه قد يبدو في غاية التعقيد لشخص لم يره من قبل.

إذا أردت أن تفهم موقف شخص لا يعرف أي شيء حول كودك، كل ما عليك فعله هو البحث عن كود لم يسبق لك قراءته ثم اقرأه. جرّب أن تفهم ماذا يفعل البرنامج كاملًا وليس فقط بعض الأسطر المنفردة، وحاول معرفة كيفيَّة إحداث تغييرات عليه عند الضرورة. هذا تمامًا ما يمر به الآخرون عند قراءتهم لكودك. ربما تستطيع الملاحظة الآن أنَّ مستوى التعقيد لا يجب أن يكون عاليًّا جدًّا حتى تصير قراءة أكواد الآخرين أمرًا محبطًا.

لذلك من الجيد أن تُخصِّص أقسامًا شبيهة بـ "هل أنت جديد على هذا الكود؟" في توثيق الكود تُقدِّم فيها بعض الشروحات التي من شأنها مساعدة الناس على فهم الكود. ويجب أن تُصاغ بحيث تكون موجهة لأشخاص لا يعرفون شيئًا حول البرنامج؛ لأنَّ الناس غالبًا ما يجهلون أي شيء حول البرنامج الذي يستخدمونه لأوَّل مرة.

الكثير من مشاريع البرمجيات انتهكت ذلك. حيث تتجه لقراءة دليل المطورين، فتُصادفك كمية هائلة من الروابط وبلا توجيه. ذلك قد يبدو بسيطًا بالنسبة لمطور قديم للمشروع؛ لأنَّ صفحة بها عدد كبير من الروابط تُمكِّنه من إيجاد الجزء الذي يبحث عنه بسرعة.

ولكن ذلك سيُعقِّد حياة الشخص الجديد على المشروع. بينما بالنسبة للمطور القديم، إضافة صفحة بأزرار كبيرة وبسيطة وإزالة قائمة الروابط تلك ستُعقِّد مهمته؛ كون هدفه الأساسي سيكون العثور على أمور مُحدَّدة بسرعة شديدة في التوثيق.

الأسوأ من التوثيق المُعقَّد غيابه، حيث يُتوقَّع منك تبيّن الأمر بنفسك أو "المعرفة المسبقة" لكيفيَّة عمل الكود. ليست تلك مشكلة للمطور؛ كون طريقة عمل برنامجه واضحة، ولكن بالنسبة للآخرين الطريقة غير واضحة بتاتًا وهذا سيُسبِّب الكثير من المشاكل.

السياق في البساطة مهم أيضًا. مثلًا، في سياق كود البرنامج، التقنيّات المُتقدمة إذا أُستخدِمَت بشكل صحيح تؤدي غالبًا إلى البساطة. ولكن تخيِّل لو أنَّ الأجزاء الداخليَّة المُتقدمة لبرنامج كهذا عُرِضَت مباشرةً على صفحة عنكبوتيّة كواجهة وحيدة للبرنامج. شي كهذا لن يكون بسيطًا في ذلك السياق، حتى بالنسبة للمطوِّر نفسه!

ما يكون أحيانًا في سياق بسيط قد لا يكون في آخر كذلك. عرض الكثير من النصوص التوضيحيّة في لوحة إعلانيَّة بجانب الطريق سيكون مُعقَّدًا للغاية؛ لعدم وجود وقت كافٍ لمرور السائقين وقراءة كل تلك النصوص، وبالتالي من الغباوة فعل هذا. بينما في دليل لبرنامج حاسوبي، إضافة الكثير من النصوص التوضيحيّة سيُبَسِّط الكثير بدلًا من كتابة جملة واحدة لوصف الشيء. ولهذا لا يوجد سطر واحد في كل فصل من الكتاب؛ لأنَّه من البساطة الزائدة قول شيء وعدم شرحه بعدها.

بالنظر في كل وجهات النظر والسياقات المُختلفة هذه، أيعني ذلك أنَّ تحقيق البساطة صعب لدرجة الاستحالة؟ لا، إطلاقًا لا يعني ذلك. يوجد لكل شيء جمهور مُستهدَف ومُحدّد، وعادةً ما يكون السياق لأي شيء فردي تفعله محصورًا للغاية. المشاكل دائمًا تُحَل، وكل ما يهم هنا هو أخذ هذه الاعتبارات في الحسبان أثناء تصميم برمجيّتك، لتبدو لِمَن يستخدمها حقًّا بسيطة.

حرب المُحرِّرات

لقد كان هناك خلافًا كبيرًا في عالم البرمجيّات بشأن الأدوات الأفضل للمهام. يحب الناس مُحرِّرات نصوص ولغات برمجة وأنظمة تشغيل مختلفة. لربما أشهر "حرب" في عالم تطوير البرمجيّات دارت بين مُستخدمين مُحرِّري النصوص: في آي (vi) وإيماكس (Emacs). يزعم مُستخدمي المُحرّرين أحيانًا أنّ مُحرِّرهم متفوِّقٌ على الآخر.

لا توجد حقيقةً أداة متفوقة على الأخرى هنا في كتابة البرمجيَّات، بل هناك أداة يراها البعض أبسط من الأخرى لأداء مهمتهم. يجد مستخدمو (إيماكس) أنَّه الأبسط لكتابة البرمجيَّات، وكذلك الأمر مع مستخدمي (في آي). يرتبط هذا لحدٍ ما باختلاف طريقة عمل أو تفكير الأفراد. لدى الناس ببساطة تفضيلات مُختلفة، ولا علاقة للصواب أو الخطأ هنا. البساطة التي يتصورها الناس في الأداة هنا ترجع للألفة، بمعنى أنَّ أي شخص يستخدم أداة مُعينة لمدة طويلة سيألفها، مما سيجعله، من وجهة نظره، يراها أبسط من أي أداةٍ أخرى. ليشعر فرد كذلك اتجاه أداة جديدة عليها أن تكون شديدة البساطة، والذي ما يكون نادرًا عادةً في مُحرِّرات النصوص البرمجيّة.

لربما يرى الغير مبرمجين أنَّ الأداتين مُعقدتين بلا سبب، الأمر الذي يُمثِّل مثالًا آخر عن نسبيّة البساطة.

يُمكن أن يكون للأدوات مشكلات تجعلها غير مُناسبة لمهمة مُعينة أو خيار خاطئ لأسباب تتعلّق بتصميم البرمجيّات (طالِع قسم "تقنيات سيئة" في الفصل الثامن). ولكن بعيدًا عن تلك المشاكل، نسبيّة بساطة الأدوات تُمكّن المُبرمجين من تحديد الأداة الأفضل لحالتهم.

كم ينبغي أن تكون بسيطًا؟

ستجول في خاطرك الكثير من الأسئلة عن البساطة عند العمل على مشاريعك. فستسأل نفسك أشياء من مثل: كم ينبغي أن أكون بسيطًا؟ كم عليّ تبسيط الأشياء؟ هل هذا بسيطٌ كفاية؟

أكيدٌ أنَّ البساطة نسبيّة كما قلنا، ولكنّها حتّى ولو كانت، فإنَّه بإمكانك أن تُنجِز بساطةً أقل أو أكثر. من منظور المُستخدِم، إمَّا أن يكون مُنتجَك سهلٌ أو صعب الاستخدام، أو حتّى بين هاتين الحالتين. وبالمثِل، من منظور مُبرمِج آخر، إمّا أن يكون كودك سهلٌ أو صعب القراءة.

فكم إذًا ينبغي أن تكون بسيطًا لتنجَح؟ كلمتان: بسيط بغباوة!

الجميل بشأن مستوى كهذا من البساطة أنَّ أي شيء إجمالًا قابل للاستخدام من الناس العاديين بهذه البساطة كذلك قابل للاستخدام من العباقرة. وبهذا تكون حصلت على مجالٍ أوسع من المُستخدمين.

ولكن لا يفهم الناس غالبًا البساطة بغباوة التي عليهم أن يكونوا عليها ليصلوا لهذا المستوى. لنأخذ مثالًا على ذلك: توجد في المجمّعات التجاريَّة خرائِط تَدُلك على الأماكن. في الخرائِط الجيدة توجد نقطة حمراء كبيرة بجانبها العبارة "أنت هنا" بأحرُف مُضخمة موضوعة أمامك لتجدها مباشرةً. بينما في الخرائط السيئة يوجد مُثلث أصفر صغير في وسط الخريطة صَعبٌ العثور عليه، وعلى الجانب كلامٌ يقول: "يُمثِّل المثلث الأصفر الصغير عبارة 'أنت هنا'". أضف هذا إلى الحيرة العامة في محاولة العثور على الأشياء عادةً في خرائط كهذه ومن الممكن أن تقضي خمس أو ست دقائق واقفًا مُتجهمًا أمام الأشياء محاولًا تبيّن كيفيَّة الذهاب إلى وجهتك.

قد يكون المثلث الأحمر ذاك بالنسبة لمُصمِّم الخريطة خيارًا معقولًا تمامًا. لقد قضى وقتًا طويلًا في تصميمه، ولهذا ذلك المثلث مهمٌ بالنسبة له كفايةً لجعله سعيدًا في قضاء خمس دقائق مُطالعًا ودارسًا أيّاه. لكن ليس هذا حالنا، أو حال الذين يرغبون فعلًا باستخدام الخريطة، فهذا شيء لن يهمنا إطلاقًا. كل ما نُريده هو أن تكون الخريطة سهلةً قدر الإمكان لنتمكَّن من إيجاد ما نريد بسرعة وننهي الأمر.

الكثـير من المـبرمجين سـيئون خاصـةً حيـال هـذا الأمـر في أكـوادهم. حيث يفترضـون أنَّ المـبرمجين الآخـرين سيسـعدون بقضاء وقتٍ طويـلٍ في دراسـة أكوادهم، لأنَّهم تعبـوا واسـتغرقوا وقتًا في كتابتهـا! الكود مُهمٌ لهم، أفلا ينبغي أن يكون مُهمًا للآخرين كذلك؟

المبرمجون عامةً أذكياء، ولكنه مازال خطأً أن تظن: "أوه سيفهم المبرمجون كُلَّ ما أكتب دون أن أوضِّح أو أشرح أيًّا من كودي". لا يهم الذكاء هنا، بل المعرفة. المبرمجون الجُدُد على كودك ولا يعرفون شيئًا عنه عليهم تعلمه، وكلما جعلت من الأسهل تعلمهم، كلما سَرُعَ فهمهم له وكلما سَهُلَ استخدامه لهم كذلك.

توجد الكثير من الطُرُق لتسهيل كودك ليتعمله الآخـرون، كإنجـاز توثيـق وتصميم بسيط وإضافة دورات تعليميّة تدريجيّة ...إلخ.

ولكن إن كان كودك ليس تعلمه سهلًا بغباوة، سيواجه الناس مشكلة معه. سيستخدمونه بشكلٍ خاطِئ، وسيسبِّبون العلل. وعندما سيحدُث كل هذا، مَنْ برأيك سَيُسأل عنه؟ صحيح، أنت! أنت من ستقضي وقتك في الإجابة عن كل أسئلتهم (أليس ذلك مُضحِكًا؟).

لا أحد منّا يُحِب أن يُستهان به أو أن يُعامل كأحمق. وهذا ما يدفعنا أحيانًا لإنشاء أمورٍ مُعقدة، لنشعر أنّه لا يُستهان بنا من المُستخدمين أو المبرمجين الآخرين. سنرمي بعدها بعض الكلمات كبيرة لشرح الكود، ونجعلها أقل من بسيطة، وعندها الناس سيحترمون عبقريتنا ولكنهم سيشعرون بالغباوة لعدم فهمهم الأمر. قد يظنون أنّنا أذكى منهم بمراحل، وذلك مُغرٍ حقيقةً، ولكن أيساعدهم ذلك حقًا؟

بينما عندما تجعل مُنتجك أو كودك بسيط بغباوة، ستُمكِّن الناس من فهمه. سيُشعرهم هذا بالذكاء، وسيمكّنهم من القيام بما يحاولون فعله، ولن يعود عليك الأمر بسوء إطلاقًا. سيُعجَب غالبًا الناس حقيقةً بك أكثر إن جعلت الأمور بسيطة من جعلها معقدة.

لا يعـني هـذا أنَّ عائلتـك سـتكون قـادرة على قـراءة كـودك. مـاتزال البسـاطة نسـبيّة، وجمهـور الكـود المُستهدف هـو المـبرمجون الآخـرون. ولكن على كـودك أن يبـدو بسـيطًا للغايـة وسـهل الفهم لجمهـورك.

يُمكنك استخدام قدر ما تحتاج من التقنيّات المُتقدمة اللازمة لتُحقِّق تلك البساطة، ولكن ما يهم أن تبقى البساطة بساطةً جوهريًّا.

عندما يلوح في خاطرك السؤال: "كم ينبغي أن أكون بسيطًا؟" سيكون عليك أيضًا أن تسأل نفسك: "أأرغب بجعل الناس تفهم هذا وتكون سعيدة أم أن يكونوا محتارين ومحبطين؟" مستوى البساطة الوحيد، إذا ما اخترت الخيار الأوَّل، الذي سيضمن نجاحك هو البساطة بغباوة.

كُن متناسقًا

يُشكِّل التناسُق جزءًا كبيرًا من البساطة. كُن مُتناسقًا؛ فإذا قُمتَ بشيء في مكانٍ بطريقة مُعيَّنة، قُم به بذات الطريقة في كُلِ مكانٍ آخر.

إذا سميّتَ مُتغيِّرًا بالشكل `somethingLikeThis` عليك تسمية البقيَّة بنفس الطريقة (مثل: `otherVariable`). إذا سميّتَ مُتغيّرًا بالشكل `named_like_this` عليك كتابة البقيّة بأحرف صغيرة ومفصولةً كلماتها بشرطة سُفليّة.

الكود الغير مُتناسِق صَعبٌ فهمه وقراءته من المُبرمجين. يُمكننا توضيح ذلك بمثال من اللغة الإنجليزيَّة. قارِن مثلًا العبارتين التاليتين:

- This is a normal sentence with normal words that everybody can understand.
- tHisisanOrmalseNtencewitHnorMalwordsthAtevErybOdycAnunderStaNd.

للعبارتان المعنى ذاته بالضبط، ولكن الأولى قراءتها أبسط بكثير من الثانية؛ كونها متناسقة مع الطريقة التي يَكتُب بها مُعظم المُتحدثين بالإنجليزيَّة عادةً. بالطبع الأولى يُمكن قراءتها كذلك، ولكن هل ستقرأ كتابًا كاملًا مكتوب بتلك الطريقة؟ بنفس المقياس، هل ستقرأ برنامجًا كاملًا مكتوبًا بلا أيِّ تناسُق؟

توجد حالات في البرمجة لا يُهم فيها الطريقة التي تفعل بها الأشياء طالما تفعلها بتناسق. يُمكنك مثلًا، نظريًّا، كتابة كود بطريقة جنونيَّة التعقيد، ولكن طالما كنت مُتناسقًا في كتابته سيجِد الناس طريقةً

لقراءته (أن يكون الكود مُتناسقًا وبسيطًا أفضل بالتأكيد، ولكن إن كان ليس بإمكانك أن تكون بسيطًا تمامًا فعلى الأقل كُن متناسقًا).

يجعل التناسُق التام كذلك البرمجة أسهل في العديد من الحالات. على سبيل المثال، لو كان كل كائِن في برنامجك يملك حقلًا يُدعى `name` سيكون بإمكانك كتابة قطعة بسيطة من الكود يُمكنها التعامل مع الحقل `name` في كل كائِنات البرنامج. بينما لو كان للكائِن `A` الحقل `a_name` وللكائِن `B` الحقل `name_of_mine` سيكون عليك كتابة جزئيّة كود خاصة للتعامل مع الكائِنين بشكل مُختلف.

ينبغي لبرنامجك أن يعمل بأسلوب مُتناسِق داخليًا كذلك. فالمُبرمِج العارِف لكيفيَّة استخدام جزءٍ من كودِك ينبغي أن يكون قادِر على معرفة كيفيَّة استخدام جزء آخر مباشرةً؛ كون الجزئين يعملان بنفس الأسلوب. فمثلًا إن كان لاستخدام الجزء `A` على المُبرمِج استدعاء ثلاث دوال وكتابة بعض الأكواد، عليه فعل المثل عندما استخدام الجزء `B`. وإن كانت هناك دالة اسمها `dump` في الجزء `A` تجعله يطبَع كل مُتغيّراته الداخلية ينبغي أن تفعل الدالة `dump` في الجزء `B` المثل. لا تُرغِم المُبرمجين على إعادة تعلُّم الطريقة التي يعمل بها نظامك في كل مرَّة ينظرون فيها إلى جزء مُختلف منه.

لربما الأمور ليست بهذا التناسق في العالم الحقيقي، ولكنَّك مسؤول عن عالم برنامجك، ويمكنك جعل الأشياء فيه بسيطة ومتناسقة كما يحلو لك.

توجد العديد من الأمثلة عن التناسق في العالم الحقيقي، وهذه إحداها: يستخدم الناس في مُعظم آسيا العيدان للأكل، بينما يستخدم الناس في أمريكا وأوروبا الأشواك. تلك طريقتان مُختلفتان للأكل، ولكن هناك تناسُق عام في كل منطقة. تخيّل الآن لو أنَّك في كل مرَّة تذهب فيها إلى بيت أحدهم عليك تعلُّم طريقة جديدة للأكل. فربما في بيت (بوب) يأكلون بالمقصات، وفي بيت (ماري) بالورق المُقوى المُسطَح. سيجعل كل ذلك الأكل مُعقدًا للغاية، أليس كذلك؟!

الشيء نفسه ينطبق على البرمجة، فبلا تناسُق كل شيء سيتعقَّد، وبه كل شيء سيَبسُط. وحتى لو كانت طريقة البرمجة ليست بسيطة، على الأقل سيكون عليك حينها تعلُّم التعقيد مرَّة واحدة وللأبد.

المقروئيّة

كما يُقال دائمًا في عالم تطوير البرمجيّات: الكود يُقرَأ أكثر مما يُكتَب. ولذلك من المهم جعل الكود سهل القراءة:

تعتمد المقروئيّة أساسًا على كَمْ المساحات المشغولة من الحروف والرموز.

إذا كان الكون بأكمله مُظلِم، ما كان ليكون بإمكانك التفريق بين الأشياء. كان كُل شيء سيبدو ككتلة واحدة. نفس الشيء مع البرامج، فإذا كان الملف كتلة من الأكواد المتكتِّلة بلا تناسُقٍ كافٍ أو مسافات منطقيّة، سيكون من الصعب جدًّا التفريق بين أجزاءه المختلفة. المسافات هي ما تُبقي الأشياء متمايزة. لست بحاجة لكتابة الكثير من المسافات؛ لأنَّه سيصبح عندها من الصعب الربط بين الأشياء، ولست بحاجة للقليل؛ لأنَّه سيصبح عندها من الصعب الفصل بين الأشياء.

لا توجد قاعدة صارمة وسريعة حول قدر التباعُد الواجِب بين الأكواد. ينبغي فقط أن يَتُم الأمر بطريقة متناسقة وكافية لمساعدة القارئ على فهم هيكلة الكود.

مثال: المسافات

من الصعب قراءة الكود أدناه لتواجد مسافات قليلة جدًّا، أي وضوح أقل لهيكلة الكود:

```
x=1+2;y=3+4;z=x+y;if(z>y+x){print"error";}
```

الكود نفسه مكتوب أدناه بكثير من المسافات بين أجزاءه معوقةً رؤية هيكلة الكود بوضوح:

```
x     =    1+   2;
y  =      +4;
 z  = x   +   y;
if (z    >   y+x)
   {    print  "error"   ;
     }
```

الكود هذه المرَّة حتّى أصعب قراءة من عندما كان خاليًا من المسافات.

وأخيرًا، أدناه نفس الكود ولكن بمسافات معقولة:

```
x = 1 + 2;
y = 3 + 4;
z = x + y;
if (z > y + x) {
    print "error";
}
```

من الأسهل قراءة الكود الموضَّح أعلاه، إضافةً إلى أنَّه يُساعد على فهم كيفيَّة تصميم المبرمج للبرنامج. هناك بدايةً إعداد لثلاث مُتغيّرات، ويليها جملة شرطيّة يُصدَر فيها خطأ. اتّضحَت هيكلة البرمجيّة للقُرَّاء بالطريقة التي استخدم بها المُبرمِج المسافات.

يُساعد جعل الكود سهل القراءة أيضًا على إصلاحه. عِندما أُستخدِمَت المسافات بشكل صحيح في مثالنا السابِق، أصبح من السهل رؤية أنَّ المتغيّر `z` لن يُصبِح مُطلقًا أكبر من `y + x` كون قيمته ستساوي دائمًا `y + x`، وبالتالي ينبغي حذف الكتلة الشرطيّة بأكملها لعدم ضرورتها.

يُمكنك عامةً إن كان كودك مليئًا بالعلل وصعب القراءة محاولة جعله أكثر مقروئيَّة بدايةً ومن ثمَّ البحث عن علله.

تسمية الأشياء

تُمثِّل تسمية المُتغيّرات والدوال والأصناف بأسماء مُعبِّرة جزءًا مُهمًا من المقروئيّة:

> **ينبغي أن تكون الأسماء طويلةً كفاية لإيصال معنى أو عمل المُسمَّى دون أن تكون طويلةً حد الصعوبة.**

من المهم أيضًا التفكير بالكيفيّة التي ستُستخدم بها هذه الدوال والمتغيِّرات عند تسميتها. فينبغي أن نسأل أنفسنا عند كتابة الأسماء بالكود: هل ستجعل هذه الأسماء الكود طويلًا لدرجة صعوبة قراءته؟ فمثلًا إن كانت لديك دالة تُستدعى مرَّة واحدة فقط في سطر لوحدها، يُمكن أن تُعطى هذه الدالة اسمًا طويلًا بلا أي مشكلة. بينما إن كنت ستستخدم الدالة مرارًا في تعبيرات مُعقدة، فعلى الأرجح أنَّه من الأفضل تقصير اسمها (إلا أنَّه ينبغي أن يكون طويلًا كفاية لإيصال ما تفعله هذه الدالة).

مثال: الأسماء

إليك كود بأسماء سيئة للغاية:

```
q = s(j, f, m);
p(q);
```

تلك الأسماء لا تُوصِل وظيفة المتغيّرات أو الدوال.

إليك الكود هذه المرَّة بأسماء جيدة:

```
quarterly_total = sum(january, february, march);
print(quarterly_total);
```

والكود نفسه أيضًا ولكن بأسماء طويلة صعبة القراءة:

```
quarterly_total_for_company_in_2011_as_of_today =
add_all_of_these_together_and_return_the_result(january_total_amo
unt, february_total_amount, march_total_amount);

send_to_screen_and_dont_wait_for_user_to_respond(quarterly_total_
for_company_in_2011_as_of_today);
```

تأخذ تلك الأسماء مسافات كثير مما يُصعِّب القراءة. هذا يوضِّح أنَّ طريقة كتابة الأسماء تؤثِّر أيضًا على المساحة المشغولة من الحروف والرموز.

التعليقات

للتعليقات الجيدة حصة كبيرة من مقروئيّة الكود. ليس عليك عامةً إضافة تعليقات لشرح عمل الكود، فعمله ينبغي أن يكون واضحًا، وإن لم يَكُن على الكود أن يُبسَّط. فقط إن كان ليس بالإمكان تبسيط الكود عندها عليك إضافة تعليق يشرح عمله.

الهدف الحقيقي من التعليقات هو شرح سبب فعلك لشيء عندما لا يكون السبب واضحًا. فإن لم يكن هناك شرحًا لذلك، سيحتار المبرمجون الآخرون، وقد يزيلون أجزاءً مهمة من الكود عندما يُغيّرونه لأنَّها لا تبدو موضوعةً لسبب.

يعتقد بعض الناس أنَّ المقروئيّة تُمثِّل كل بساطة الكود، أي أنَّ بجعلك الكود سهل القراءة تكون قد أنهيت عملك كمُصمِم. هذا ليس صحيحًا، فحتى لو كان الكود سهل المقروئيَّة يُمكن أن يظل نظامك البرمجي مُعقدًا للغاية. جعل الكود مقروءًا أمرٌ مُهم، وهو عادةً أوَّل خطوة على طريق تصميم برمجيَّة جيدة، ولكنَّه ليس كل شيء.

البساطة تتطلَّب تصميم

لا يبني الناس بطبيعة الحال للأسف أنظمةً بسيطة. يتحوَّل النظام بدون الاهتمام بالتصميم إلى وحش ضخم ومُعقَّد.

إذا ما كان مشروعك يفتقد لتصميمٍ جيد، ومع استمرار نموه، سيُصبِح في نهاية المطاف أكثر تعقيدًا مما تتحمَّل.صعبٌ تخيُّل هذا عند البعض، وخاصةً الذين لا يستطيعون تخيُّل مُستقبَل بعد إطلاق المشروع، والذين لا يملكون الخبرة الكافية لفهم مقدار التعقيد المُمكن أن تصل إليه الأشياء. هناك كذلك ثقافة شائعة تقول: "بدأنا توًّا العمل على ميزات جديدة، فعلينا إنجاز الأمور بالطريقة الصحيحة، إلا أنَّنا لا نستطيع وذلك لأنَّ بلا بلا بلا بلا". في يومٍ من الأيام، باتباع ثقافة كتلك، سيفشل مشروعك. ولن يَهُم حينها أسباب فشله، لأنَّ ذلك لن يُغيّر حقيقة أنَّه فَشِل.

بعيدًا عن ذلك، عندما تُصمِّم عملًا جيدًا، غالبًا لن تُشكَر كثيرًا. الفَشَل الذريع في التَصميم يُلاحَظ، بينما التغييرات الصغيرة في العمل التي تؤدي إلى تصميمٍ جيد ليست مرئيَّة للذين ليسوا على صلةٍ مباشرة بالكود. هذا قد يجعَل عمل المُصمِّم صعبًا، فالتعامل مع الإخفاقات الكبيرة يَجلُب لك شُكرًا كبيرًا، ولكن منع واحدة من الحدوث يُحتمَل أن لا يُلاحظه أحد.

دعنا نَشكُركَ هنا نيابةً عن الجميع. هل اهتممتَ قليلًا بالتصميم؟ رائع وشكرًا جزيلًا لك! سيَحصد مُستخدميك وزُملائك فوائد هذا (والتي تتمثَّل ببرمجيّة تعمل وإصدارات بموعدها وقاعدة أكواد واضحة ومفهومة) وستشعر بالراحة إزاء عملك وستذهَب إلى منزِلك شاعِرًا بالإنجاز. أسيعرِف المطورون الآخرون مقدار العمل الذي استغرقه جعل الأمور تعمل بهذه السلاسة؟ ربما لا، ولكن لا بأس في هذا. هناك أمور مُجزية أخرى غير تهاني أقرانِك.

قد يُقدِّر البعض نادرًا عملك. لا تيأس، فأحدهم سيُلاحظ في النهاية. ولحينها استمتع بالنتائِج الإيجابيَّة لتصميمك الفعّال والصحيح.

قد يأخذ المبرمجون المبتدئون أو زملائك وقتًا طويلًا، عندما يرونَك تُطبِّق مبادئ التصميم في هذا الكتاب على عملك، ليفهموا لمَ عليهم التصميم جيدًا أيضًا. جعلهم يقرؤون الكتاب سيُساعِد، أو استمر بإرشادِهم (أو إجبارِهم في أسوأ الأحوال)، إذا لم يكونوا قادرين أو راغبين بقراءته، على اتخاذ قرارات تصميميّة صحيحة. قد يأخذهم الأمر بضع سنوات (كحدٍ أقصى) ليروا كيف أنَّ القرارات التصميميّة الجيدة تؤتي ثمارها، ولكنهم سيفعلوا.

الفصل الثامن: التعقيد

عندما تعمل كمبرمج محترف، من المحتمل أنَّك ستعرف ذلك الشخص (أو أنك أنت ذلك الشخص!) الذي يمر بهذه القصة الرهيبة لتطوير مشروع مشترك: "بدأنا العمل على هذا المشروع منذ خمس سنوات، والتقنية التي كنَّا نطورها كانت حديثة وقتها، لكنها باتَتْ قديمة الآن. وبدأت الأمور مع هذه التقنية البالية تزيد تعقيدًا أكثر فأكثر. وبَدَوْنا وكأننا لن ننهي هذا المشروع أبدًا. ولكن إذا أعدنا كتابة المشروع من جديد، فسوف نجلس عليه خمس سنوات أخرى!".

قصة شائِعة أخرى: "لا نستطيع أن نطوّر المشروع بسرعة كافية تُبقيه ضمن احتياجات المستخدم الجديدة"، أو: "بينما كُنَّا نُطوِّر المشروع، طوَّرَت الشركة الفلانيِّة مُنتَجًا أفضل من منتجنا وأسرع منّا بمراحل".

بِتنا نعلم الآن أنَّ مصدر كل هذه المشاكل هو "التّعْقيد". تَبْدُأ في البداية بمشروع بسيط يمكن إنهاؤه خلال شهر واحد، بعد ذلك تُضيف تعقيدات وتصبح المهمة تحتاج لثلاث أشهر. ثم تَأخذ كل جزء من هذا المشروع على حدة وتجعله أكثر تعقيدًا، فتُصبح مدة المشروع تسع أشهر.

تعقيد فوق تعقيد، لا يحدث الأمر بشكل خطّي. لذلك لا يمكنك أن تفترض افتراض مثل: "مشروعي يحتوى على عشرة ميّزات، وإضافة ميّزة واحدة أخرى لن تزيد مدة العمل إلّا 10%". ولكنَّك في الحقيقة سوف تُضطّر إلى تنسيق هذه الميزة مع كل الميزات العشرة الموجودة. لذلك إن كانت هذه الميزة ستستغرق عشرة ساعات لكتابة كودها، فسوف تستغرق عشرة ساعات أخرى لجعلها متفاعلة مع الميزات العشرة الأخرى. كلما زاد عدد الميزات الموجودة زادت تكلفة إضافة ميزة جديدة. يمكنك تقليص حجم هذه المشكلة عبر جَعْلِ تصميمِ برمجياتك ممتازًا، ولكن ستبقى هناك بعض التكاليف الخفيفة لكل ميزة جديدة دائمًا.

تبـدأ بعض المشـاريع بمجموعـة معقَّدة من المتطلبـات والـتي لن يتمكَّن مطوِّروهـا من إصـدار النسـخة الأولى منها أبدًا. إذا كانت هذه حالتك فعليك أن تتخلَّى عن بعض الميِّزات. لاتحاول صنع الأفضل وكل ما تتمنَّاه في نسختك الأولى، بل اصنع شيئًا يعمل فقط ثم حسِّنه مع الوقت.

توجد طُرُق أخرى لزيادة التعقيد غير طريقة إضافة ميِّزات جديدة، وأكثرها شيوعًا:

- **توسيع غاية البرمجيّة**

لا تفعل عمومًا هذا أبدًا. قد يكون قسم التسويق لديك متحمّسين جدًّا لِفكرة إنشاء برمجيَّة واحدة تدفع ضرائبك وتحضّر لك وجبة العشاء، لكن عليك أن ترفض كل اقتراح يشابه هذا بكل ما لديك من قوّة كلّما طُرِح عليك. استمر بغاية برمجيِّتك الحاليّة. كل ما تحتاجه هو أن تقوم هذه البرمجيّة بوظيفتها جيّدًا، وسوف تنجح بهذه الطريقة (طالما كانت برمجيّتك تساعد الناس بأمور يحتاجون المساعدة فيها).

- **ضَمُّ مبرمجين إضافيين**

نعم، هذا صحيح. إضافة أشخاص جُدُد إلى الفريق لا تُبسّط الأمور، بل على العكس من ذلك، ستزيد الأمور تعقيدًا. هناك كتاب مشهور بعنوان "أسطورة رجل بالشهر" كتبه (فريد بروكس) أشار فيه إلى هذه النقطة. إذا كان لديك عشر مبرمجين، فإنَّ إضافة المبرمج الحادي عشر تعني إضاعة وقت بينما يتأقلم هذا المبرمج، وإضاعة وقت حتى يتأقلم المبرمجون العشر مع المبرمج الجديد، وأيضًا إضاعة وقت في تفاعل المبرمج الجديد مع المبرمجين العشر السابقين، وهكذا دواليك. لديك فرصة نجاح أكبر مع فريق عمل مُكوَّن من عدد قليل من المبرمجين المحترفين مقارنةً بفريق مُكوَّن من عدد كبير من المبرمجين الغير محترفين.

- **تغيير الأشياء التي لا تحتاج إلى تغيير**

كلّما تُجري تغييرًا جديدًا فأنت بذلك تزيد التّعقيد، سواءً أكان هذا التغيير متطلّبات أم تصميم أو حتى جزء من الكود. أنت بِهذا تزيد من احتماليَّة حدوث العلل البرمجيَّة، كما أنَّك تزيد الوقت اللازم لاتخاذ قرار بشأن التغيير، وتزيد أيضًا الوقت اللازم لتنفيذ هذا التغيير، والوقت اللازم للتحقُّق من أنَّ التغيير الجديد يعمل مع باقي أجزاء البرمجيَّة، والوقت اللازم لتتبُّع التغيير، والوقت اللازم لاختباره. كل تغيير يعتمد على الآخر في ظل كل هذا التّعقيد، لذلك كلّما أجريت تغيير كلّما زاد الوقت الذي سيستغرقه كل تغيير جديد. مازال من المهم إجراء بعض التغييرات الحَتْميَّة، ولكن علينا أن نتَّخذ قرارات مدروسة بشأن هذه التغييرات، وليس إجراءها بهوانا.

- **التقيّد بالتقنيات سيئة**

هذا يحدث أساسًا عندما تستخدم تقنية ما وتستمر باستخدامها لمدة طويلة لأنك تعتمد عليها كثيرًا. هذه التقنية في هذه الحال تُعتبَر سيئة إذا قيّدتك (أي أنّها لا تُمكّنك من الانتقال لتقنية أخرى بيُسْر في المستقبل)، أو إذا كانت ليست بهذه المرونة التي تُلبّي احتياجاتك المستقبلية، أو حتّى إن كانت لا توفّر لك تلك الجودة التي تحتاجها لتصميم برمجيّة بسيطة.

- **سوء الفهم**

المبرمجون الذين ليس لديهم فهم كافٍ حول عملهم سيطوّرون أنظمة معقّدة، وقد يؤدّي هذا إلى حلقة مفرغة: سوء الفهم سيؤدّي إلى التّعقيد، والذي بدوره سوف يؤدّي إلى فهم أسوء، وهكذا. واحدة من أفضل الطرق التي تُنَمّي من خلالها مهاراتك التصميمية هي أن تتأكّد من أنك قد فهمت النظام والأدوات التي تعمل معها بشكل جيّد. كلّما فهمتهم بشكل أفضل، وكلّما عرفتَ أكثر حول البرمجيّة، كلّما كان تصميمك أبسط.

- **غياب التصميم أو ضعفه**

هذا يعني أساسًا "فشل في التخطيط للتغيير". الأشياء ستتغيّر، وتصميم العمل مطلوب للحفاظ على البساطة أثناء نمو المشروع. عليك تصميم النظام جيدًا في البداية والاستمرار في التصميم جيدًا مع توسُّع النظام، وإلا ستُدخِل التعقيد بسرعة إلى كودك؛ لأنَّ في التصميم السيء كُل ميزة تُضاعِف تعقيد الكود بدلًا من إضافة القليل منه وحسب.

- **إعادة اختراع العجلة**

إذا اخترعتَ مثلًا ميثاقًا خاصًا بك رغم وجود واحد مُمتاز، ستُمضي وقتًا طويلًا بالعمل على هذا الميثاق بدلًا من قضائه في العمل على برمجيّتك. لا ينبغي أبدًا أن تخترع اعتماديَّة ضخمة لعملك (كخادم عنكبوتي أو ميثاق أو مكتبة ضخمة) ما لم تَكُن هي مُنتجَك نفسه. الحالات الوحيدة التي يكون فيها إعادة اختراع العجلة مقبولة هي الحالات التالية:

1. تحتاج لشيء غير موجود.
2. "العجلات" الموجودة تقنيات سيئة ستُقيّدُك.
3. "العجلات" الموجودة غير قادرة أساسًا على تلبية احتياجاتك.
4. "العجلات" الموجودة غير مُصانة جيدًا ولا قدرة لك على تولي صيانتها (لأنَّك مثلًا لا تمتلك كودها المصدري).

كل هذه العوامل مُؤذية ببطئ وتدرُّج لمشروعك، وليست فورية الضرر. معظمها ستؤدي فقط لضرر بعيد المدى - ضرر لن تُدرِكه لسنةٍ أو أكثر - ولذلك إذا ما اقترحها أحد ستبدو عديمة الضرر. وفي كل مرَّة تُنفِّذ إحداها ستبدو الأمور بخير، ولكن مع مرور الوقت، وخاصةً عندما تتكدَّس الأمور، سيغدو التعقيد أكثر وضوحًا وسيزداد نموُّه شيئًا فشيئًا، حتى تُصبِح ضحية أخرى لقصة الرعب: "منتَج لم يُشحَن".

التعقيد والغاية

ينبغي أن تكون الغاية الأساسيّة لأي نظام شديدة البساطة، ما يُساعد على إبقاء النظام كاملًا بالبساطة المُمكنة واقعيًّا. ولكن إن بدأت بإضافة ميزات تُلبي غايات أخرى ستزداد الأمور تعقيدًا بسرعة. فمثلًا، الهدف الرئيسي لمعالج النصوص مساعدتك على كتابة الأشياء. إذا جعلناه فُجاءةً قادرًا على قراءة البريد الإلكتروني، ستزداد الأمور تعقيدًا بسخافة. أيمكنك أن تتخيّل كيف ستبدو واجهة المُستخدِم؟ أين سنضَع كل الأزرار؟ يُمكننا القول أنَّ هذا انتهاكًا لغاية مُعالِج النصوص. أنت لم تُوسّع حتّى غايته، بل أضفت ميزة لا علاقة للبرنامج فيها.

من المهم كذلك التفكير بغاية المُستخدِم. سيُحاوِل مُستخدِمك القيام بشيء ما، وغاية برنامجك ينبغي أن تكون قريبة جدًّا لغايته. لنقل مثلًا أنَّ غاية مُستخدِم دفع ضرائبه. سيرغَب ذلك المُستخدِم ببرمجيَّة غايتها مُساعدة الناس على دفع ضرائبهم.

سيُصعِّب عدم توافُق غايتك مع غاية المُستخدِم حياته. فمثلًا لن تتفق مستخدم غايته قراءة بريده مع البرنامج الذي يستخدمه التي غايته عرض الإعلانات للمُستخدمين.

تريد رؤية المُستخدِم يغضب بسرعة شديدة؟ اجعل من الصعب عليه إنجاز غايته. افتح نوافذ مُنبثقة بوجهه عندما يحاول القيام بشي، وأضف الكثير من الميزات للبرنامج حتى لا يُمكنه إيجاد الميزة التي يُريد، واستخدم الكثير من الأيقونات الغريبة التي لا يُمكنه فهمها. هناك العديد من الطرق لفعل هذا، ولكن كُلها تؤدي إلى التضارُب مع غاية المُستخدِم أو إلى انتهاك الغاية الأساسيّة للبرنامج نفسه.

يستخدم المسوِّقون أو المدراء أحيانًا أهدافًا لا تتوافق مع الغاية الأساسية للبرنامج، كاستخدامهم: "أصبِح جذابًا" أو "احصل على تصميم عصري" أو "أضحِ مشهورًا مع وسائِل الإعلام" أو "استخدم آخر التقنيات" وغيرها من العبارات الرنانة. قد يكون هؤلاء مُهمين في مؤسستك، ولكنهم بالتأكيد ليسوا من ينبغي أن يُقرِّر ما على البرنامج فعله! عملك كمُصمِّم برمجيات أو مُدير تقني أن تُراقِب بقاء البرنامج على

مسار عمله وأن لا تُنتهَك غايته الأساسيّة. لا أحد غيرك سيتحمَّل هذه المسؤوليّة. أحيانًا قد يكون عليك حتى الكفاح لأداءها، ولكنها تستحق حقًا على المدى البعيد.

وليس بالضرورة أن تفشَل تسويقيًّا بسبب هذه الفلسفة، فهناك الكثير والكثير من المُنتجات التي كانت شديدة النجاح بالرغم من أنَّها مُحتفظة بغايتها وحسب. فمثلًا الصابونة غايتها تنظيف الأشياء، والملح تمليحها، والمصباح الكهربائي إنارتها. كل تلك المنتجات، بالرغم من قيامها بغايتها الأساسيّة وحسب، دُعِمَت من شركات ضخمة لعقود. ليس عليك تصميم مُنتَجًا مُعقَّدًا لتنجَح تسويقيًّا، عليك فحسب أن تملك معرفة ومهارات تسويقيّة والذي هو بالأمر البعيد تمامًا عن مجال تصميم البرمجيّات.

لا أحد سيرغب حقًا بالحصول على برنامج خيالي ومُعقَّد والقيام بخمسمئة شيء من خلاله. سيكون المُستخدمون أسعد بمُنتَج بسيط وُمرَّكز الغاية التي لا ينتهكها أبدًا.

تقنيات سيئة

اختيار التقنيات الخاطئة في نظامك هو مصدر آخر شائِع للتعقيد، وخاصةً تلك التقنيات التي لا تستطيع تلبية متطلبات مشروعك المستقبليّة بكفاءة. سيكون من الصعب معرفة، بدون التنبؤ بالمستقبل، أي تقنيَّة عليك اختيارها الآن. توجد، ولحسن الحظ، ثلاث عوامل يُمكنك النظر إليها لتحديد ما إذا كانت التقنيّة "سيئة" قبل البدء باستخدامها حتى، وهي: **احتماليّة الدوام** و**التوافقيّة** و**الاهتمام بالجودة**.

احتماليّة الدوام

إنّ احتماليّة دوام تقنية هي احتماليّة أن تبقى هذه التقنية مدعومة. إذا تورّطتَ بمكتبة ما أو بعض الاعتماديّات التي أصبحت قديمة وغير مدعومة، فأنت واقع في بعض المشاكل حقًّا.

بإمكانك أن تحصل على فكرة عامّة حول احتماليَّة دوام برمجيَّة ما عن طريق الاطّلاع على تاريخ آخر إصدار منها. هل يُصدِر مطوّروها نُسخًا جديدة تَحُلّ مشاكل المستخدم بانتظام؟ وما مدى استجابة المطوّرين لتقارير العلل البرمجية المُبلّغ عنها؟ هل لديهم قائمة بريديَّة أو فريق دعم نَشِط؟ هل يتحدّث أشخاص كُثُر عن هذه التقنية في الشابكة؟ إذا كان هناك الكثير من الزّخَم حول هذه التقنية الآن فيمكنك التأكّد إلى حدٍ ما أنَّها لن تموت قريبًا.

وانظر أيضًا ما إذا كان يُحرِّكُها مورِد واحد فقط أم أنَّها مُستخدمة على نطاق واسع وفي العديد من مجالات البرمجيَّات وبواسطة مُخْتَلَف المطوّرين. إذا كان مورِد واحد هو الذي يُحرّك ويُوجِّه النظام، فهناك مخاطر من أن يُفلِس هذا المورِد أو حتّى أن يُوقِف دعمه للنظام وحسب.

شعبيّة التقنية

ربّما قد يبدو وكأننا نقول أنَّ عليك أنْ تقتني التقنية الأكثر شعبيّة من التي تناسب احتياجاتك، وهذا صحيح إلى حدٍّ ما، فالتقنيات الشعبيّة لديها احتماليَّة دوام كبيرة، ولكن عليك أن تُفرِّق بين الأدوات الشعبيّة حقًّا وبين الأدوات التي اكتسبت شعبيّتها بسبب سياسة منتجيها الاحتكاريّة.

أثناء كتابة هذا الكتاب، تُعتَبَر لغة البرمجة سي (C) أحد الأمثلة على الشعبيّة الحقيقيَّة. الكثير من الناس يستخدمونها في الكثير من المُنظّمات المختلفة ولأسباب مختلفة. إنّها موضوع العديد من المعايير العالميّة، وهناك عدّة تنفيذات لهذه المعايير، من ضمنها العديد من المُصرِّفات الرّائجة.

التّوافقيّة

التّوافقيّة هي مقياس لمقدار سهولة الانتقال من التقنية في حال اضطررت لذلك. لكي تأخذ فكرة حول توافقيّة تقنية ما، اسأل نفسك: "هل يمكنني التعامل مع هذه التقنية بطريقة معيارية تمكّنني من الانتقال لنظام آخر يتبع نفس المعيار؟". توجد مثلًا معايير عالميَّة لكيفيَّة تفاعل البرنامج مع أنظمة قواعد البيانات. تدعم بعض قواعد البيانات هذه المعايير جيّدًا. فإذا اخترت أحد أنظمة قواعد البيانات الجيِّدة تلك؛ فسيكون من السهل عليك في المستقبل أن تنتقل إلى نظام قاعدة بيانات آخر مع القليل من التغييرات الطفيفة في برنامجك.

ولكن بعض أنظمة قواعد البيانات الأخرى لا تدعم هذه المعايير بشكل جيّد. فإذا كنت تريد أن تتنقل بين أنظمة قواعد البيانات التي لا تدعم المعايير، فعليك أن تعيد كتابة برنامجك من جديد. لذلك عندما تختار أحد تلك الأنظمة اللامعياريَّة، فأنت بذلك أصبحت مُقيَّد بها ولن يكون بإمكانك الانتقال لنظام مختلف بسهولة.

6. قد يكون بعض المطوّرين عاطفيّين جدًّا حيال التقنية التي يستعملونها، ولتجنّب الإساءة إلى أي مستخدم لم تُذكَر أي تقنية هنا.

الاهتمام بالجودة

هذا العامل يُعتبَر مقياسًا شخصيًّا أكثر، لكن الفكرة هي متابعة ما إذا حُسِّنَ المنتج في الإصدارات الأخيرة. إذا كنت تستطيع الاطّلاع على الكود المصدري، فتأكد ما إذا كان المطوّرون يعيدون بناء ويطوّرون قاعدة الأكواد. هل تُصبح أكثر سهولةً للاستخدام أم أكثر تعقيدًا؟ هل يهتمّ المسؤولون عن صيانة التقنية بجودة منتجاتهم؟ هل كانت هناك مؤخرًا الكثير من الثغرات الأمنيّة الخطيرة في البرمجيّة الناتجة عن برمجة سيئة؟

أسباب أخرى

هناك جوانب أخرى يجب مراعاتها عند اختيار تقنية، وبالذات بساطتها ومدى ملائمتها لغاياتك. يمكن للآراء الشخصية أن تلعب في ذلك دورًا أيضًا بعد أن تأخذ جميع الاعتبارات العملية بعين الاعتبار. بعض الناس يفضّلون طريقة لغة برمجة والتي تبدو أفضل من طريقة لغة أخرى، وهذا سبب صحيح لاختيار تقنية أحيانًا. إذا كنت تفضّل تقنية على أخرى، وكانت كل الصفات الأخرى متطابقة، فاختر تلك التي تجعلك سعيدًا. فبالنّهاية أنت الذي ستستخدمها، ورأيك مهم. ستساعدك الإرشادات المذكورة أعلاه على التخلُّص من الخيارات السيئة تمامًا، والباقي يعتمد على أبحاثك الشخصية ومتطلباتك ورغباتك.

التّعقيد والحل الخاطئ

إذا كانت الأمور تؤول إلى تعقيد كبير، فهذا يعني عادةً أنَّ هناك خطأ في تصميم البرمجيّة في مرحلة تسبق المرحلة التي ظهر فيها التعقيد.

فمثلًا من الصّعب جدًّا أن تسير السيّارة إذا كانت عجلاتها مربّعة الشكل. ضبط المُحرِّك لن يحلّ المشكلة هنا، عليك أن تُعيد تصميم السيارة بعجلات دائريَّة.

كلّما وجدتَ "تعقيد لا يُحَل" فهذا بسبب خطأ في أساس التصميم. إذا بَدَت المشكلة لا تُحَل في مرحلة، ارجع إلى المراحل السابقة لعلّك تجد سببها.

كثيرًا ما يفعل المبرمجون هذا، فقد تجد نفسك قائلًا: "لدي هذا الكود الفوضوي جدًّا، وإضافة ميِّزة جديدة أمر شديد التعقيد!". مشكلتك الأساسيّة هنا هي أنَّ الكود فوضوي. رتِّبه بدايةً واجعله أبسط وستجد أنَّ إضافة ميِّزة جديدة أصبحت أمرًا بسيطًا أيضًا.

ما المشكلة التي تحاول حلّها؟

إذا سألك أحدهم: "كيف أجعل هذا المُهر يطير إلى القمر؟" فسيكون سؤال هذا: "ما هي المشكلة التي تحاول حلّها؟". قد تجد أنّ ما يحتاج هذا الشخص لفعله حقًّا هو جمع بعض الأحجار الرماديَّة عوضًا عن الذهاب للقمر. لماذا اعتقدَ أنّ عليه أن يطير إلى القمر وأن يستعين بمُهر لفعل هذا؟ هو من قد يعرف ذلك فقط. يحصل لدى النّاس ارتباك كهذا، اسألهم عن المشكلة التي يحاولون حلّها وسيبدأ الحل بالظهور. فمثلًا، في الحالة السابقة، عندما تفهم المشكلة بالكامل سيكون الحل بسيط وواضح: كل ما على هذا الشخص فعله هو أن يتمشى في الخارج قليلًا ليجد بعض الأحجار الرماديَّة، ولن يحتاج عندها لمُهرٍ أو لقمر.

لذلك عندما تتعقَّد الأمور ارجع واطّلع على المشكلة التي تحاول حلّها. ارجع خطوة كبيرة للخلف، كل الأسئلة مسموح بها. ربّما اعتقدت أنَّ الطريقة الوحيدة للحصول على 4 هي جمع 2 مع 2 ولم يخطر على بالك أن تجمع 1 مع 3، أو أن تضع 4 مباشرةً دون الحاجة لعمليَّة الجمع. أي طريقة تحل هذه المشكلة (مشكلة "كيف أحصل على العدد 4؟") مقبولة، لذلك كل ما عليك فعله هو أن تجد الطريقة الأفضل لحالتك التي أنت بها.

تجاهل افتراضاتك، وتمعّن في المشكلة التي تحاول حلّها، وتأكد من أنَّك فهمت كل جانب فيها ثمّ أوجد الطريقة الأبسط لحلّها. لا تسأل: "كيف أحل المشكلة بالكود الحالي؟" أو "ما هي الطريقة التي حلّت

الأستاذة (آن) بها المشكلة التي في برنامجها؟". وهكذا قد ترى كيف يجب إعادة صياغة الكود، ثم ستتمكّن من إعادة صياغته، ثم ستتمكّن من حل المشكلة.

المشاكل المعقّدة

قد تُستدعى أحيانًا لحل مشاكل معقّدة أصلًا، مثل التدقيق الإملائي أو برمجة حاسوب ليلعب الشطرنج، وهذا لا يعني أنَّ حلك يجب أن يكون معقّدًا، بل يعني أنَّ عليك العمل بجهد أكبر من العادة لتبسيط الكود مع برامج كهذه.

إذا كنت تواجه صعوبة مع مشكلة معقدة، فاكتبها على ورق بلغة واضحة، أو ارسمها كرسومات تخطيطية. أفضل البرامج تتم على الورق أحيانًا وإدخالها إلى الحاسوب مجرد خطوة بسيطة.

يُمكِن حل أصعب المشاكل التصميميّة عبر رسمها أو كتابتها ببساطة على ورقة.

معالجة التعقيد

سيواجهك الكثير من التعقيد أثناء عملك كمُبرمِج؛ من مُبرمجين كتبوا أنظمة مُعقدة عليك إصلاحها، إلى مُصممي لغات وعتاد صعَّبوا عليك عملك.

هناك طريقة مُحدَّدة لإصلاح الأجزاء المُعقدة من نظامِك: أعِد تصميم الأجزاء الفرديَّة من النظام بخطوات صغيرة ومُتدرجة. ينبغي أن يكون كل إصلاح صغيرًا بقدر ما يُمكنك إجراءه بأمان دون إدخال المزيد من التعقيد. الخطر الأكبر الذي ستواجِهه أثناء قيامك بهذا هو خطر إمكانيَّة إدخالك لمزيد من التعقيد بالإصلاحات التي تُجريها، ولهذا تفشل العديد من عمليات إعادة التصميم والكتابة كُليًّا؛ حيث تُدخِل تعقيدًا أكثر من الذي تُصلِحه، أو تُخرِج نهايةً نظامًا مُعقدًا كما كان في الأصل.

يُمكِن أن تكون كل خطوة صغيرة كصِغَر إعطاء مُتغيّر اسمًا أفضل، أو كصِغَر إضافة بضعة تعليقات إلى كودٍ مُحيّر. ولكن غالبًا ما تنطوي الخطوات على فَصِل جزءٍ مُعقَّد إلى عدّة أجزاء أبسَط.

لنقَل مثلًا أنَّك تملك ملفًا طويلًا يحتوي كل كودِك؛ ابدأ بتحسينه عبر فَصِل كل جزء صغير إلى ملف مُستقِل، ومن ثمَّ تحسين تصميم ذلك الجزء. افعل المِثل مع باقي الأجزاء حتى تحصَل نهايةً على نظامٍ مفهومٍ وفعّالٍ ويُصَان.

سيستغرقك الأمر عملًا أكثر إن كان النظام شديد التعقيد، ولهذا عليك أن تكون صبورًا. ابدأ بالتفكير بنظامٍ أبسط من الذي لديك، حتى ولو بطريقةٍ صغيرة، ومن ثمَّ نفِّذه تدريجيًّا. حالما تحصل على ذلك النظام الأبسط أعِد الكرَّة؛ فكِّر بنظامٍ أبسط واعمل على تنفيذه. لا تُفكِر بنظامٍ "كامل"، فلا شيء كامل. حاول فقط العمل على شيء أفضل مما لديك، لتحصل نهايةً على قدرٍ عالٍ من البساطة.

تجدر الإشارة إلى أنَّه لا يُمكنك التوقف عن كتابة ميِّزات جديدة وقضاء كل وقتك على إعادة التصميم. يُخبرنا قانون التغيير أنَّ بيئة برنامجك ستتغيّر باستمرار، ولهذا على وظائِف برنامجك أن تتلاءم معها. سيُكبِّدك فشل مُلاءمتك وتحسينك للبرنامج من منظور المُستخدِم لمدّة طويلة خطر خسارة مُستخدميك وخطر احتضار مشروعك.

توجد عدّة طُرُق، لحُسن الحظ، للموازنة ما بين الحاجة لكتابة ميِّزات جديدة ومعالجة التعقيد. إحدى أفضل الطُرُق هي أن تُعيد التصميم بهدف تسهيل تنفيذ ميزة مُعينة (ومن ثمَّ تنفيذها بالطبع). وبهذه الطريقة ستتنقل بانتظام ما بين العمل على إعادة التصميم وتنفيذ الميزات. سيُساعد هذا كذلك على جعل تصميمك الجديد يُلائِم احتياجاتك جيدًا؛ كونك تُنشئه واضعًا استخدام له في عين الاعتبار. وبهذا سيغدو نظامك أقل تعقيدًا مع الوقت، وستستمر بتلبية حاجات مُستخدميك. يُمكنك حتى فِعل المِثِل مع العلل؛ بحيث إذا وجدتَ أنَّ علَّة ما يُمكن تسهيل إصلاحها بتصميم بديل، أعِد تصميم الكود قبل إصلاحها.

إعادة التصميم لتنفيذ ميزة

يُخزِّن مشروع يُدعى (بجزيلا) بياناته في قاعدة بيانات. يدعم (بجزيلا) نوع واحد من أنظمة قواعد البيانات يُدعى OldDB. أراد بعض الزبائِن الجدد استخدام نظام قواعد بيانات مُختلف يُدعى NewDB. كان لدى هؤلاء الزبائِن أسباب جيدة تدفعهم لطلب هذه الميزة: حيث كانوا يفهمون النظام NewDB أكثر بكثير من OldDB، وكانوا يستخدمون النظام NewDB مُسبقًا في شركاتهم. كل الزبائِن الآخرين كانوا مُستمرين باستخدام النظام القديم.

ولذلك كان على (بجزيلا) دعم استخدام أكثر من نوع من قواعد البيانات. كان هذا سيتطلب تغييرات كبيرة في الكود، وخاصةً أنَّ (بجزيلا) كانت لا تملك أي كُود مركزي لتخزين واستقبال المعلومات من قاعدة البيانات، بل كان يتم ذلك باستخدام عدد كبير من أوامر قواعد البيانات المُخصَّصة للنظام OldDB فقط والمنتشرة في الكود والتي لن تعمل مع النظام الجديد.

أحد الخيارات التي طُرِحت هي نثر عبارات شرطيَّة في ملفات المشروع، تُنفِّذ أكوادًا مختلفة مُخصَّصة لكل قاعدة بيانات فيهم. كان هذا الخيار ليُضاعِف تعقيد قاعدة الأكواد كاملةً، كما أنَّ فريق (بجزيلا) كلُّه مؤلَّف من القليل من المطورين بدوام جزئي. فمضاعفة تعقيد النظام كانت ستؤدي إلى عدم القدرة على صيانته.

قرَّر فريق (بجزيلا)، عوضًا عن ذلك، إعادة تصميم النظام لتسهيل دعمه لعدّة قواعد بيانات. إليك نظرة سريعة على الكيفيَّة التي أنجزوا فيها هذا المشروع الضخم:

1. توجد بعض الأوامر المعياريَّة التي تعمل على أي نوع من أنظمة قواعد البيانات، ولكنَّها غير مُستخدمة دائمًا في الكود الحالي. مرّوا على أكواد النظام كاملًا وأصلحوا كل ملف ليستخدم الأوامر المعياريَّة حيث أمكَن.

2. أنشؤوا للأوامر التي ليس لها نسخة معياريَّة دوالًا تُعيد الأمر الصحيح لقاعدة البيانات المُستخدمة، وبدلوا كل أمر غير معياري باستدعاء دالته المقابلة في النظام.
3. توقفوا عن استخدام الميزات الخاصة التي تعمل على قاعدة البيانات القديمة فقط، واستبدلوا تلك الميزات بأخرى مقابلة معياريَّة تعمل على جميع قواعد البيانات.
4. أعادوا تصميم نظام تثبيت (بجزيلا) ليُنصِّبَ نفسه على أي قاعدة بيانات كانت وليس على القديمة فقط. انطوت هذه العملية على إعادة تصميم نظام التثبيت أولًا لتبسيطه، ومن ثمَّ ملاءمته ليدعم قاعدتي البيانات.

شكَّلت كل خطوة من الخطوات السابقة مشروعًا بحدِّ ذاتها، وقُسِّمَت كلٌّ منها إلى خطواتٍ أصغر؛ بهدف التصميم جيدًا في كل جزء من العمل. أُختُبِرَ النظام أيضًا بعد كُل تغييرٍ أُجري؛ للتأكُّد من استمرار عمله كما كان يعمل سابقًا مع قاعد البيانات القديمة.

أكانت نتيجة كل ذلك نظامًا كاملًا؟ لا، ولكن النتيجة كانت نظامًا أفضل من السابِق. فإلى جانِب أنَّه أصبح يدعم قاعدة البيانات NewDB، أصبحت كذلك صيانته أسهل. توسَّع مشروع (بجزيلا) في نهاية المطاف ليدعم أربع أنظمة قواعد بيانات مختلفة، وكل ذلك بفضل هذا العمل الذي سهَّل دعم قواعد بيانات جديدة[7].

7. أُعيدَ تصميم مشروع (بجزيلا) عدّة مرَّات بنفس الطريقة على مرّ السنوات لأسباب عديدة ومختلفة. يُمكنك مطالِعة سجل الأعمال الرئيسيّة التي تمَّت من هنا، أو مطالعة السجل المُخصَّص بكيفيَّة إنجاز دعم قواعد البيانات من هنا. ستُعطيك قراءة عناوين العناصر فكرة عامة عن الكيفيَّة التي أُنجِز بها المشروع إذا كنت مُلِمًّا بأنظمة قواعد البيانات.

تبسيط الأجزاء

كان ذلك كلامًا رائعًا، ولكن كيف سنُبسّط هذه الأجزاء حقًا؟ هنا حيث تلعب المعرفة حول تصميم البرمجيّات دورها. ستساعدك دراسة الأنماط التصميميّة المختلفة وطُرُق التعامل مع الأكواد القديمة وكل أدوات هندسة البرمجيّات عمومًا. سيكون من المفيد خاصةً معرفة عدّة لغات برمجة والإلمام بعدّة مكتبات مُختلفة؛ كونه لكلٍ منها طُرُقها المختلفة في التعامل مع المُشكلات، والتي قد يكون من المُمكن تطبيقها على حالتك، حتى وإن لم تكن تستخدم هذه اللغات أو المكتبات.

ستُعطيك دراسة تلك المواد خيارات عديدة لتختار منها عندما تُواجه تعقيدًا معُينًا. تعاونك قوانين تصميم البرمجيّات على اختيار الخيارات الجيدة، ومن ثمَّ حكمك وخبرتك هي من تُحدِّد الخيار الأنسب لمشكلتك. لا تستخدم مُطلقًا أداة آليًا لأنَّ سُلطة ما تعتبرها الأفضل، وإنما افعل دومًا الأفضل للكود الذي تعمل عليه والوضع الذي أنت فيه.

قد يُصادِف أحيانًا أن لا تعرِف أي أداة تستخدم لتبسيط كودك، أو قد تكون مُبرمِجًا جديدًا وليس لديك الوقت الكافي لدراسة كل هذه المعلومات حالًا. عليك في حالات كهذه أن تنظر إلى التعقيد الذي تواجهه وتسأل نفسك: "كيف يُمكِن تسهيل التعامل مع هذا أو جعله مفهومًا أكثر؟". هذا هو السؤال المفتاحي خلف أي تبسيط. أي إجابة صحيحة على هذا السؤال هي طريقة صالحة لتبسيط كودك. أدوات وتقنيات تصميم البرمجيّات ما هي إلا طريقة لمساعدتنا على استنتاج إجابات أفضل لهذا السؤال.

التعقيد الذي لا يُصلَح

قد تواجهك بعض التعقيدات صعبٌ تجنبها عند عملك على تبسيط نظامك، كتعقيد البنية العتاديّة المُستخدَمة. ينبغي أن يكون هدفك في حالات كهذه إخفاء التعقيد. غلّف ذلك التعقيد بطريقة تُسهّل على المبرمجين الآخرين فهمه واستخدامه.

إعادة الكتّابة

يرمي بعض المصممين النظام كاملًا عندما يبلغ شدّة التعقيد ويبدؤون من الصفر. إعادة كتاب النظام من الصفر أساسًا اعتراف بفشل المُصمِّم، فهو كمَن يقول: "لقد فشلنا بتصميم نظامٍ يُصان ولذلك علينا البدء من الصفر مجدّدًا".

يعتقد البعض أنَّ كل الأنظمة ستُعاد كتابتها يومًا ما. هذا ليس صحيحًا، فمن المُمكِن كتابة نظام لن يُرمى مطلقًا. مُصمِّم البرمجيّات الذي يقول: "سيكون علينا رمي كل شيء يومًا ما على أيّة حال" هو كالمُهندِس المعماري الذي يقول: "ستسقط ناطحة السحاب هذه يومًا ما على أيَّة حال". من المؤكَّد أنَّ ناطحة السحاب ستسقط إن كانت سيئة التصميم وغير مُصانة جيدًا. بينما إن كانت مبنيَّة جيدًا من البداية ومُصانة بحرص، فلِمَ علّها تسقط؟

من المُمكِن بناء أنظمة برمجيِّة تُصان كما من المُمكِن بناء ناطحات سحاب كذلك.

تكون هناك حالات بالرغم من هذا فيها إعادة الكتابة مقبولة، إلا أنَّها قليلةٌ جدًّا. ليس عليك إعادة كتابة نظامك إلا إن كانت كل الحالات التالية مُحقَّقة:

1. قدَّرتَ بدقة أنَّ إعادة كتابة النظام ستكون أقل هدرًا للوقت من إعادة تصميم واحد موجود. لا تخمِّن الأمر وحسب، وإنما أجرِ تجارب بإعادة تصميم النظام الموجود لترى كيف ستجري الأمور. قد يكون من الصعب جدًّا مواجهة تعقيد موجود وحل بعض أجزائه، ولكن عليك محاولة ذلك فعليًّا عدّة مرَّات قبل أن تستطيع معرفة كم الجهد اللازم لإصلاح التعقيد كاملًا.
2. لديك الكثير من الوقت لتقضيه في إنشاء نظام جديد.
3. أنت بقدرٍ ما مُصمِّمٌ أفضل من مُصمِّم النظام الأصلي، أو أنَّ مهاراتك، إن كنت المُصمِّم الأصلي، تحسَّنَت جذريًّا منذ ذلك الحين.

4. اعتزمتَ تمامًا تصميم النظام الجديد بسلسلة من الخطوات البسيطة ولديك مُستخدمين جاهزين لمنحك تعليقاتُهم في كل خطوة.
5. لديكَ الموارد المتاحة لصيانة النظام الموجود ولتصميم النظام الجديد بنفس الوقت. لا تتوقف مطلقًا عن صيانة نظام لديه مُستخدمين ليُعيدَ المبرمجون كتابته. ينبغي أن تُصان الأنظمة دومًا طالما كانت مُستخدمة. وتذكَّر أنَّ اهتمامك الشخصي أيضًا مورِد ينبغي أن يُؤخَذ بعين الاعتبار، فهل لديك وقتًا كافيًا كل يوم لتكون مُصمِّمًا للنظام الجديد والقديم؟

قد تكون في موقِفٍ تكون فيه إعادة الكتابة أمرًا مقبولًا فقط إن كانت كل النقاط السابقة مُحقَّقة، وإلا الشيء الصحيح الذي سيكون عليك القيام به هو معالجة تعقيد النظام الموجود دون إعادة كتابته، عبر تحسين تصميم النظام بسلسلة من الخطوات البسيطة.

الفصل التاسِع: الاختبار

لا يُمكِن التيقُّن من أنَّ البرنامج سيعمل مُستقبلًا، كل ما يُمكن التيقُّن منه أنَّه يعمل الآن. حتى وإن شغلته مرَّة، قد لا يعمل التي بعدها. لربما بسبب تغييرات في البيئة حالِت إلى عدم عمله، أو لربما شغلته في حاسوبٍ مُختلِف فلم يعمل عليه.

هناك رغمًا من هذا أملًا لعمله، وقانون الاختبار يُخبرنا بماهيّته:

درجة معرفتك لسلوك برمجيّتك هي درجة دقة اختبارك لها.

كلما اختبرت برمجيّتك حديثًا، كلما زادت احتماليّة استمرار عملها حاليًا. وكلما كَثُرَت البيئات التي تختبر برمجيّتك فيها، كلما زادت احتمالية عملها في تلك الحالات. هذا الجزء الذي عنيناه عندما قلنا "درجة الاختبار"، أي عدد جوانب البرنامج التي اختبرتها حديثًا وعدد البيئات المُختلفة التي اختبرتها فيها. يُمكن تبسيط هذا عامةً بقول:

أنت لا تَعرِف ما إذا كان يعمل ما لم تختبره.

"يعمل" قولٌ مُبهم، فما الذي نعنيه؟ ما تعلمه عندما تختبر برمجيِّتك أنَّها ستَسلُك كما أردتها أن تفعل. وبالتالي عليك معرفة كيف تُريدها أن تَسلُك. قد يبدو هذا غبيًّا ومُبهمًا، ولكنَّه من الحقائق المُهمة في الاختبار. عليك أن تسأل سؤالًا دقيقًا في كُل اختبار، وأن تحصَل على إجابةٍ دقيقة. فمثلًا عليك أن تسأل: "ماذا سيَحدُث عندما يضغط المُستخدِم على هذا الزر كأول شيء يفعله بعد بدء البرنامج لأوَّل مرَّة؟"، ويُمكن أن يكون الجواب كهذا: "سيعرض التطبيق نافذة تقول 'أهلًا بالعالم!'".

والآن حصلت على سؤالك وأصبحت على عِلِم بماهيّة الجواب. حصولك على أجوبة أخرى في الاختبار يعني أنَّ برمجيّتك "لا تعمل".

يكون أحيانًا اختبار السلوك صعبًا، ويكون السؤال عامًا مثل: "هل سينهار البرنامج إذا فعل المُستخدِم هذا؟" وسيكون الجواب الذي تتوقعه "لا". ولكن ستحصل في البرمجيَّات حَسنة المصميم، في معظم الحالات، على معلومات أكثر دقة من تلك في اختباراتك.

عليك كذلك أن تجعل اختباراتك دقيقة. فإذا أنتجَت اختباراتك أن البرنامج يَسلُك جيدًا بينما هو ليس كذلك، أو إذا أعطتك أنَّه مُعطَّل بينما هو ليس كذلك، فمعنى هذا أنَّها ليست بالاختبارات الدقيقة.

عليك أخيرًا أن تُتابِع نتائِج اختباراتك لتتأكَّد أنَّها صحيحة. فإذا ما فَشِلَت، عليك أن تكون قادرًا على معرفة سبب وكيفيَّة فشلها بدقّة.

من السهل نسيان الاختبار. ستكتب بعض الأكواد ثم تحفظها وتنسى أن تنظر ما إن كانت تعمل. لا يهم مدى عبقريتك كمبرمج، ولا كم الإثباتات الرياضيَّة التي تُظهِر أنَّ كودك سليمًا، لن تعلَم أنَّه يعمل ما لم تُجرِّب استخدامه.

وفي أي مرَّة تُغيِّر فيها جزئيَّة من البرمجيَّة لن تكون قادرًا على التيقُّن من أنَّها تعمل بعد الآن. عليك مُجدّدًا اختبارها. قد تكون كذلك هذه الجزئيَّة مُرتبطة بجزئيّات أخرى، مما يؤدي إلى عدم تيقُّنك من عمل أيٍ من تلك الجزئيّات أيضًا. قد تحتاج حتى لاختبار البرنامج كاملًا إن كان تغييرك كبيرًا.

أنت بالتأكيد لا تَرغَب باختبار برنامجك كاملًا يدويًّا في كُل مرَّة تُجري فيها تغييرًا صغيرًا. ولهذا، في هذه الأيام، يُطبِّق المطورون عادةً هذا القانون عبر إنشاء اختبارات مُؤتمتة لكل جزئيَّة كود يكتبونها. الشيء جميل حيال هذا أنَّهم بإمكانهم تشغيل الاختبارات بعد كل تغيير يُجرونه مباشرةً لتَختبِر، هذه الاختبارات المؤتمتة، كل جزء من النظام للتأكُّد من أنَّ كل شيء مازال سليمًا بعد التغييرات.

هناك الكثير من المعلومات في الشابكة والكتب عن كتابة الاختبارات المؤتمتة والاختبار عامةً. يشرح قانون الاختبار فقط السبب والوقت الذي علينا الاختبار فيه، والمعلومات التي تمنحنا أيّاها هذه الاختبارات.

الملحق أ: قوانين تصميم البرمجيّات

يُلخِّص هذا المُلحَق كل القوانين التي تطرقنا لها في هذا الكتاب:

1. الغاية من تصميم البرمجيّات مُساعدة الناس.

2. معادلة تصميم البرمجيّات:

$$\text{ر} = \frac{\text{ق}_{\text{آ}} + \text{ق}_{\text{م}}}{\text{ج}_{\text{ت}} + \text{ج}_{\text{ص}}}$$

حيث أنَّ:

- ر يُمثِّل مقدار رغبة القيام بالتغيير.
- $\text{ق}_{\text{آ}}$ يُمثِّل القيمة الآنيّة.
- $\text{ق}_{\text{م}}$ يُمثِّل القيمة المستقبليّة.
- $\text{ج}_{\text{ت}}$ يُمثِّل الجهد التنفيذي.
- $\text{ج}_{\text{ص}}$ يُمثِّل جهد الصيانة.

يُمكِن اختزال مُعادلة تصميم البرمجيّات الأساسيّة الموضَّحة أعلاه مع مرور الوقت إلى:

$$\text{ر} = \frac{\text{ق}_{\text{م}}}{\text{ج}_{\text{ص}}}$$

ما يُبيّن أنَّ تخفيض جهد الصيانة أكثر أهميّة من تخفيض الجهد التنفيذي.

3. قانون التغيير: كلما طالت مُدّة وجود برنامجك، كلما زادت احتماليّة وجوب تغيير أيّ جزءٍ منه مستقبلًا.

4. قانون احتماليّة العيب: تتناسب فرصة أن تنتج عيبًا في برنامجك طردًا مع حجم التغييرات التي تجريها عليه.
5. قانون البساطة: تتناسب سهولة صيانة أي جزئيّة من برمجيّتك مع بساطة أجزائِها الفرديّة.
6. قانون الاختبار: درجة معرفتك لسلوك برمجيّتك هي درجة دقة اختبارك لها.

هذا كل ما في جعبتنا لهذا الملحق. هناك المزيد من الحقائِق والأفكار الأخرى في هذا الكتاب، ولكن هذه الست عناصر هي قوانين تصميم البرمجيّات. لاحظ أنَّ من بين هذه الست عناصر الأكثر أهميَّة أن تؤخذ في عين الاعتبار: غاية البرمجيّات والصيغة المختزلة من معادلة تصميم البرمجيّات وقانون البساطة.

إذا أردت تلخيص الحقائق الأهم عن تصميم البرمجيّات لتَذكُرِها في جملتين بسيطتين يُمكنك القول:

- تخفيض جهد الصيانة أكثر أهميّة من تخفيض الجهد التنفيذي.
- يتناسب جهد الصيانة مع تعقيد النظام.

مُسلَّحًا بهاتين العبارتين وبفهم لغاية البرمجيَّات، بإمكانك إعادة تطوير علم تصميم البرمجيّات بالكامل، بشرط أن تفهم أيضًا أنَّ تعقيد النظام يأتي من تعقيد أجزاءه الفرديّة.

الملحق ب: حقائق وقوانين وقواعد وتعريفات

يسرد هذا الملحق كل حقيقة وقانون وقاعدة وتعريف أساسي تطرقنا له في هذا الكتاب:

- حقيقة: يَكمُن الفرق بين المبرمج الجيد والمبرمج السيء في الفهم. حيث أنَّ المبرمج السيء لا يستوعب جيدًا ما يقوم به، على نقيض المبرمج الجيد.
- قاعدة: يجب على المبرمج الجيد أن يبذل كل ما بوسعه ليُبسِّط ما يكتبه للمبرمجين الآخرين.
- تعريف: البرنامج هو:
  1. سلسلة من التعليمات المُمرَّرة للحاسوب.
  2. الإجراءات التي يتخذها الحاسوب كنتيجة للتعليمات المُمرَّرة.
- تعريف: يندرج أي شيء يتعلق بمعمارية نظامك أو القرارات التقنية التي تقوم بها أثناء إنشائها ضمن فئة: "تصميم البرمجيات".
- حقيقة: كل مَنْ يكتب البرمجيّات يُعدّ مُصمِّمًا.
- قاعدة: لا ينبغي أن تكون عمليّة التصميم عمليَّة ديموقراطيَّة، وينبغي أن تُتّخَذ القرارات فرديًّا.
- حقيقة: هناك قوانين يُمكِن ويمكنك معرفتها. هذه القوانين أبدية وغير متغيرة وصحيحة في جوهرها وتعمل.
- **قانون**: الغاية من تصميم البرمجيّات مُساعدة الناس.
- حقيقة: أهداف تصميم البرمجيَّات:
  1. تمكيننا من كتابة برمجيّات مفيدة قدر الإمكان.
  2. تمكين برمجياتنا من الاستمرار في أن تكون مفيدة قدر الإمكان.
  3. تصميم برمجيَّات يُمكِن إنشاؤها وصيانتها بيُسِر وسهولة قدر الإمكان بواسطة مُبرمجيها، وبالتالي يُمكن أن تكون - وأن تستمر في أن تكون - مُفيدة قدر الإمكان.

- **قانون**: معادلة تصميم البرمجيَّات:

$$\text{ر} = \frac{\text{ق}_{\text{آ}} + \text{ق}_{\text{م}}}{\text{ج}_{\text{ت}} + \text{ج}_{\text{ص}}}$$

عند ترجمتها لغويًّا تُصبِح:

تتناسب الرغبة بإجراء تغيير طردًا مع القيمة العائدة مِنه وعكسيًّا مع مقدار الجهد المبذول لإجراءه.

تُختزَل المعادلة مع مرور الوقت لتصبح:

$$\text{ر} = \frac{\text{ق}_{\text{م}}}{\text{ج}_{\text{ص}}}$$

ما يُبيّن أنَّ تخفيض جهد الصيانة أكثر أهميّة من تخفيض الجهد التنفيذي.

- قاعدة: ينبغي أن تكون جودة تصميمك متناسبة مع المدى الزمني المُستقبلي الذي سيستمر نظامك في مساعدة الناس خلاله.
- قاعدة: هناك بعض الأشياء لا يمكنك التنبؤ بها حول المُستقبَل.
- حقيقة: إحدى أكثر الأخطاء التي يَقع فيها المبرمجون كارثيَّةً وشيوعًا هي محاولة التنبؤ بشيء مُستقبلي متجاهلين حقيقة أنَّهم لا يمكنهم ذلك.
- قاعدة: أنت أأمَن إذا لم تحاول التنبؤ بالمُستقبل بتاتًا. بل اتخاذ، عوضًا عن ذلك، قراراتك بناءً على المعلومات المباشرة المعروفة في الزمن الحاضر.
- **قانون**: قانون التغيير: كلما طالت مُدّة وجود برنامجك، كلما زادت احتماليّة وجوب تغيير أيّ جزءٍ منه مستقبلًا.

- حقيقة: الأخطاء الثلاث التي يميل مصمموا البرمجيَّات للوقوع فيها أثناء تعاملهم مع "قانون التغيير":
  1. كتابة كود لا حاجة له.
  2. عدم الاهتمام بجعل الكود سهل التغيير.
  3. الكتابة بعموميّة مبالَغ فيها.
- قاعدة: لا تكتب كودًا حتّى تحتاج فعليًّا إليه، واحذف أي كود غير مُستخدَم.
- قاعدة: ينبغي أن يُصمَّم الكود بناءً على ما تعرفه الآن، وليس على ما تظن أنَّه سيحدُث مُستقبلًا.
- حقيقة: إذا كان تصميمك يُعقِّد الأشياء عوضًا عن تبسيطها، فأنت تُهندِس بفواقة.
- قاعدة: كُن عامًّا بقدر ما تحتاج أن تكون الآن.
- قاعدة: يُمكنك تجنُّب الأخطاء الثلاث باتباع مبدأ التصميم والتطوير التصاعدي.
- **قانون**: قانون احتمالية العيب: تتناسب فرصة أن تنتج عيبًا في برنامجك طردًا مع حجم التغييرات التي تجريها عليه.
- قاعدة: أفضل تصميم هو ذاك الذي يسمح بأكبر تغيير في البيئة مع أقل تغيير في البرمجيَّة.
- قاعدة: لا "تُصلِح" أي شيء ما لم يكن مُشكلةً، ولا تفعل حتى تملك دليل يؤكِّد أنَّه كذلك.
- قاعدة: لا ينبغي أن تُضاف أي قطعة من المعلومات، في أي نظامٍ كان، مرَّتين.
- **قانون**: قانون البساطة: تتناسب سهولة صيانة أي جزئيَّة من برمجيّتك مع بساطة أجزائِها الفرديّة.
- حقيقة: البساطة نسبيّة.
- قاعدة: كُن بسيطًا بغباوة إن أردت الظَفَر بالنجاح.
- قاعدة: كن مُتناسقًا.
- قاعدة: تعتمد المقروئيّة أساسًا على كَمْ المساحات المشغولة من الحروف والرموز.

- قاعدة: ينبغي أن تكون الأسماء طويلةً كفاية لإيصال معنى أو عمل المُسمَّى دون أن تكون طويلةً حد الصعوبة.
- قاعدة: ينبغي أن تشرح التعليقات سبب فعل الكود لشيء لا ماهيّة عمله.
- قاعدة: البساطة تتطلَّب تصميم.
- قاعدة: يمكنك زيادة التعقيد عبر:
  - توسيع غاية البرمجيّة
  - ضَمُّ مبرمجين إضافيين
  - تغيير الأشياء التي لا تحتاج إلى تغيير
  - التقيّد بالتقنيات سيئة
  - سوء الفهم
  - غياب التصميم أو ضعفه
  - إعادة اختراع العجلة
  - انتهاك غاية برمجيِّتك
- قاعدة: يُمكنك تحديد ما إذا كانت برمجيّة "سيئة" بالنظر إلى: احتماليَّة دوامها وتوافقيّتها والاهتمام بجودتها.
- قاعدة: إذا تعقَّد شيء فغالبًا هناك خطأ في التصميم في مكانٍ ما في مرحلة سابقة لمرحلة ظهور التعقيد.
- قاعدة: عندما تواجِه تعقيدًا، اسأل: "ما المشكلة التي تحاول حلّها؟".
- قاعدة: يُمكِن حل أصعب المشاكل التصميميّة عبر رسمها أو كتابتها ببساطة على ورقة.
- قاعدة: لمعالجة تعقيد في نظامك، أعِد تصميم أجزاءه الفرديّة بخطوات صغيرة.

- حقيقة: السؤال المفتاحي خلف كل التبسيطات الممكنة: "كيف يُمكِن تسهيل التعامل مع هذا أو جعله مفهومًا أكثر؟".
- قاعدة: إذا ما واجهتَ تعقيدًا يقبَع خارج برمجيِّتك، غلّفه بطريقة تُبسِّطه للآخرين.
- قاعدة: إعادة الكتابة مقبولة فقط في مجموعة محدودة للغاية من الحالات.
- **قانون**: قانون الاختبار: درجة معرفتك لسلوك برمجيّتك هي درجة دقة اختبارك لها.
- قاعدة: لن تعلَم أنَّه يعمل ما لم تُجرِبه.